8086マシン語秘伝の書
■日高徹／青山学 著
啓学出版

8086マシン語秘伝の書
日高徹／青山学 著
啓学出版

8086マシン語秘伝の書

■日高徹／青山学 著

啓学出版

はじめに

コンピュータ言語にも色々なものがありますが、ひとつだけすべてに共通していることがあります。それは、最終的には必ずマシン語になって、あるいはマシン語によって動いているという事実です。

フランス語、英語、スペイン語、日本語……。言葉に種類があるように、マシン語にもいくつかの種類があります。通常、ニモニックによるアセンブリ言語もマシン語といいますが、Z 80、8080、8086，6502、6809、80286、80386 ……など、CPUの違いによってマシン語（ニモニック）はバラバラです。さらに、人間の言葉に方言があるように、マシン語にはハードウェアの違いという方言以上に面倒な壁が存在しています。

そのため、同じCPUを搭載したコンピュータでも、実際の用法は機種によってまったく違うものとなります。その結果、CPU別にマシン語の解説をしようとすると、どうしてもニモニックそのものの解説が中心になりがちです。また、機種別にマシン語を解説しようとすると、今度はハードウェア・コントロールに関する説明が必須となり、プログラムは急激に現実味を帯びたものになってしまいます。

そもそも、マシン語というものは複数の命令によって意味をなす言語ですから、ニモニックは単なるアルファベット的な存在にすぎません。したがって、本来ならば基本的なアルゴリズムや基礎テクニックをマスターしてから、現実的なプログラムへと進むべきものです。しかし、それではマシン語に対する興味を持続することが困難となるので、とりあえずゲームなど楽しいことを目標にマシン語を覚えるわけです。

そこで、たとえ順序は逆であっても、ある程度マシン語をマスターしたなら、改めてゆっくりと基礎テクニックを振り返ってほしいのです。なぜなら、ハードウェアをコントロールすることはマシン語のほんの一端であり、その前後のプロセスにこそマシン語の真のテクニックがあるからです。

マシン語プログラムを組む人の中には、プログラムの無駄やアルゴリズムなど、どうせアセンブルすれば隠れてしまうという人もいます。これは、マシン語の基礎を省いてしまったことの弊害かもしれません。せっかく覚えたマシン語ですから、面白くないこともそれなりに知っておくことが大切なのです。

とはいえ、基礎テクニックが基礎のままではヤッパリ面白くありません。そこで、マンガを読むような気楽な気分で、つまらない基礎を知識の一部にしてしまいましょう。本書の基礎テクニックは、いわばプログラムの化粧品のようなもの。ちょっぴりプログラムにオシャレをするだけで、プログラムが見違えるほど光り輝くかもしれません。

いかにも機械的なマシン語ですが、アルゴリズムやテクニックというものには芸術に匹敵する'美'が存在しています。それを追求するかしないか、そのこと自体はプログラムを組む人の自由です。しかし、マシン語の技術レベルの差とは、案外そんなところに潜んでいるのではないでしょうか。

1990年9月　　日高　徹
青山　学

目次

一の章　**秘伝の書を読むまえに** ―――― *1*

二の章　**秘伝の問答86** ―――― *5*

問1	セグメントの概念	芭蕉夫・輪田智（スイス）	*6*
問2	マイナスの数値	初心者マー君（埼玉）	*9*
問3	慣用句的論理演算	少年アセンブラ（岐阜）	*11*
問4	レジスタの0チェック	W大商学部の星（ブライトン）	*13*
問5	「LEA」の効用	LEA不信のJRマニア（神奈川）	*15*
問6	RAMの特徴を活かす	どすこいドカ弁（香川）	*17*
問7	FARタイプのプロシージャ	メンチ（芸能界）	*20*
問8	無駄命令とは	人間無駄蒸気（秋田）	*22*
問9	ESレジスタの利用	ボケない老人（栃木）	*25*
問10	「XCHG ES,DX」を実現	夜ふかしヒグマ（北海道）	*27*
問11	セグメント・レジスタの初期化	魔法使い見習いジロー（魔法の国）	*29*
問12	「EXX」命令を実現	星降る乙女（与論島）	*31*
問13	INC命令とキャリーフラグ1	悩める美人のOL（福岡）	*33*
問14	＋／－を交互に	マシン西郷（鹿児島）	*35*
問15	MOV命令を減らす	浪花ぼてじゃこ物語（大阪）	*38*
問16	ビット転送	Donald Tack（ロサンゼルス）	*40*
問17	プロシージャへのジャンプ	ドラQ（新潟）	*42*
問18	命令の書き換え	ペンネーム猫目漱石（愛媛）	*45*
問19	フラグで合図を送る	下宿先のドラQII（山形）	*48*
問20	範囲のある比較	ルートカ将軍（メソポチャ星）	*51*
問21	レジスタペアの値を2倍に	武蔵旅情（玄海灘）	*53*
問22	マイナスの値を1/2にする	ツバメ小次郎（巌流島）	*55*
問23	かけ算を分解して高速化その1	難破船長パフェ（太平洋）	*58*
問24	かけ算を分解して高速化その2	エスパー魔脳（宮崎）	*60*
問25	CLDとSTD	ニューハーフ占い師マーサ（東京）	*63*
問26	CMP命令でのジャンプ	ホワイトエンジェル（佐賀）	*66*
問27	テーブルを利用したジャンプ	クラブ小話（和歌山）	*68*
問28	一度のCMP命令で5つの条件判断	サンタ日本代理人・黒須三太（富士山）	*70*

問 29	共通項のあるジャンプ	中年ベビー（兵庫）	73
問 30	スタックを利用したジャンプ	半魚人（沖の鳥島）	75
問 31	ビット別のジャンプ	喫茶シェルブール（長崎）	77
問 32	フラグ以外の条件ジャンプ	岩内保羅夫（マ界党）	79
問 33	LOOP命令の特徴	パチンコ老人（愛知）	81
問 34	INC命令とキャリーフラグ 2	現地名・早ロトム（ニューヨーク）	84
問 35	基礎的な疑似乱数	ペンギン野郎（南極）	86
問 36	相加法による乱数	宝塚九二男（兵庫）	89
問 37	乱数利用の基礎	男一匹長州軍団（山口）	91
問 38	乱数の応用	旅先にて……風船のトラ（福井）	94
問 39	乱数に変化をつける	中 3 あきな（石川）	97
問 40	BPレジスタとは	悩める牧師（ニューギニア）	100
問 41	パラメータ渡しプログラム・コード編	天才バーボン（酒乱界）	104
問 42	ブロック充填	名物・金脈おじさん（佐渡島）	106
問 43	データ転送	根津見狐造（住所不定）	109
問 44	データ高速転送	怪盗ルビィの指輪ちゃん（広島）	111
問 45	3バイトの加減算	青い珊瑚礁（沖縄）	114
問 46	ストリング命令のセグメント・オーバーライド	卑弥呼の部屋（奈良）	116
問 47	十六進数を十進数に	番長ボロ屋敷（宮城）	118
問 48	BCD数値の加減算	カリブの海賊（ネズミーランド）	120
問 49	データの左右反転	投稿マニア（徳島）	123
問 50	実用できる左右反転プログラム	ピータン国王（岩手）	125
問 51	AFフラグとは	閻魔大王クッパ（地獄一丁目）	127
問 52	「CMP DS,ES」を実現	明治キメラ（長野）	129
問 53	「CMP DX:AX,＊＊＊」を実現	ニセ舞妓（京都）	131
問 54	レジスタペアのNEG命令	ヒッチくん（エジンバラ）	133
問 55	ジャンプ代わりにリターン	白夜の棋士（静岡）	135
問 56	ビット情報の縦方向コピー	眠りプログラマー（福島）	137
問 57	ブロック比較について	ゲゲゲのQ太郎（正マ界）	139
問 58	SCASB命令の効果的用法	ドブねずみ男（邪マ界）	142
問 59	SCASB命令とテーブル	目ン玉おやじ（本マ界）	145
問 60	ベスト5のチェック	村長連合（大分）	148
問 61	1バイト数値ソート	徳ノ川秀麿（島根）	150
問 62	2バイト数値ソート	田舎教師（茨城）	152
問 63	ブロック単位の文字列サーチ	貧乏神（貧民峡谷）	154
問 64	連続データからの文字列サーチ	ましむご大僧正（魔心寺）	156

問 65　ブロック単位の文字列ソート　トマス・エジさん（富山）　159
問 66　シフトJISコードからJISコードへ　ピーコー物語（熊本）　161
問 67　アスキーコードのかけ算（筆算タイプ）　マシン語歌人・俵まり（岡山）　163
問 68　4バイトのかけ算（筆算タイプ）　ばぐちゃん（メモリの森）　167
問 69　AX÷CX＝BX（小数第一位七捨八入）　旅情の人（ベネチア）　169
問 70　アスキーコードの割り算（小数第一位七捨八入）　影丸（三重）　171
問 71　BX÷CX＝BX.AL（小数第三位七捨八入）　サスケ（滋賀）　173
問 72　BX.AL×BP＝AX（小数第一位七捨八入）　総領の甚六（高知）　176
問 73　アスキーコードからBXレジスタへ変換　アストロ・ベイダー（TWS星）　178
問 74　アスキーコードからBCDの数値へ変換　技術部長ノホホン（TRS星）　181
問 75　BCDの数値をアスキーコードへ変換　銀河のシェリフ（地球出身）　184
問 76　BCDをシフトする　ドコデモガイジン（千葉）　186
問 77　BCD数値の四捨五入　スライムちゃん（テレビ界）　189
問 78　BCD数値×AL　セブンっ子（鳥取）　191
問 79　BCDどうしのかけ算（筆算タイプ）　ワシは父ちゃん（群馬）　193
問 80　BXレジスタの値をBCDに変換　デタラム・ペテム（エジプト）　198
問 81　マシン語コード分割　西洋の魔女（ノートルダム）　202
問 82　BCDどうしの割り算（割る数の桁数固定）　敷島博士（山梨）　204
問 83　内部割り込みと外部割り込み　天馬天之助（天国）　208
問 84　MAKEの利用　マシン語迷人（人間界）　210
問 85　BCDに関するミニ・テクニック　夢見る夢（夢脳）　213
問 86　テーブル処理で複雑な計算を　さすらい人（礼文島）　216

三の章　**正しいマシン語のために** ―――――――― 221

8086 ニモニック表
アスキーコード一覧表

一の章
秘伝の書を読むまえに

本書は、世界各地から……いや存在さえ定かでない謎の世界からも寄せられた多くの質問の中から、8086の秘伝として後世に残すべき基礎テクニックばかりを集め、それに応える形で本にまとめ上げたマシン語の極意書であります。

その昔、日本には忍者という超能力にも似た秘術を使いこなす人間がいました。もちろん、忍者は一朝一夕になれるようなものではありません。常に死と背中合わせの厳しい修行を重ね、その中から自分の天分に見合った忍法を体得した者だけが、忍者として一流の称号を得ることができたのです。いくら厳しい修行に耐えようとも、オリジナルの忍法を開発できなかった者は、結局は三流の雑魚忍群として虫けらのような一生を送るしかなかったのです。

しかし、一流の忍者にも老化という力の衰える時期が必ずやってきます。その時、彼らは自分の秘術を残すために、その極意を巻物に記したといいます。こうして、秘術は巻物とともに後継者に受け継がれていくのです。もちろん後継者になれるのは、弟子の中から選ばれた、たった一人の超一流の忍者です。

……やがて、彼もその巻物に秘術を記す時が訪れます。こういった小さな歴史が代々繰り返されることにより、巻物はその流派の『秘伝忍法帖』、あるいは『秘伝の書』として無上の存在価値を持つようになるのです。つまり、「その巻物さえ手に入れれば、無条件で超一流の忍者になれる」という神話が出来上がるわけです。

こうなると、忍者であればその巻物が欲しくなって当然です。巻物を求めて、密告、裏切りが横行し、ついには疑心暗鬼から内ゲバへと発展、修行を忘れた凄惨な争いが起こるようになるのです。その結果、多くの忍者の里が闇から闇へと自滅していったのです。たったひとつの巻物のために……。

それほどまでに人の心を狂わす『秘伝の書』……。その内容は、実は秘術を記したものではなかったのです。そこには、精神的な極意、つまり平常心を養うことの大切さや秘訣が抽象的に記されていただけでした。

忍の道を極めた者にとって、それこそがあらゆる忍法の極意であり、そこか

ら完成された秘術など一代限りのどうでもいい枝葉末節のことだったのです。厳しい修行はそれを悟るための手段であり、秘伝とはそういった悟りの境地を伝えることなのです。

すなわち、『秘伝の書』の価値はそれに見合う修行を積んだ者にのみ理解され、修行中の者には文面通りの内容しか伝わらないのです。例えば、部屋にある敷居の上は誰でも歩くことができますが、千尋の谷の上に架けられた同じ幅の木の上を平然と歩くことができるでしょうか。誰にでもできる簡単なことを、環境を選ばず冷静に実行できるようになるには、たゆまぬ基礎訓練が必要なのです。その中から自分の才能に適した新しいものを発見した時、オリジナルの秘術が生まれるのです。そして、その精神を平易に教えてくれるのが『秘伝の書』というわけです。

マシン語でプログラムを組めるようになると、どうしてもアッと驚く秘術ばかりを開発したくなります。しかし、秘術の陰には基礎テクニックがあるのです。あくまでも、秘術はその延長線上にあることを忘れないでください。

本書にある基礎テクニックは、まさにマシン語のための『秘伝の書』となるべき空気のような存在です。あるいは、「なんだ、こんなもの!!」と思うかもしれません。基礎とはそんなものです。しかし、つまらないことを当り前に使えるか使えないかで、プログラムは大きく変わってくるのです。本書の内容がつまらなく思えなくなった時、オリジナルの秘術的マシン語プログラムが完成する時といえるでしょう……。

なお、本書は 8086 系(8086,80286,80386･･･)をCPUとするコンピュータ用のマシン語プログラムを、MS-DOSのMASM上で開発することを前提に書かれています。したがって、プログラムやテクニックを実用化するためには、MASMを使用するための基本知識が必要です。

ニモニックやアセンブラの必要性がわからない場合は、まずマシン語の基礎を教えてくれるような本を読んでください。本書では、マシン語を覚えるための基礎は学ぶことができません。あくまでも、「マシン語テクニックの基礎をマシン語を使って学ぶ」ということが目的です。

また、お手持ちのアセンブラによっては、二進数が使用できないとか色々な

制限があることがあります。そういった入門用のアセンブラをお使いの方は、できるだけ早いうちに新しいものを手に入れることを考えたほうがいいでしょう。少なくとも、本書のプログラム程度は問題なくアセンブルできるようなものでないと、マシン語で秘術を凝らすには荷が重いかもしれません。まずは、アセンブラを空気のような存在にすること、それがマシン語の最初の基礎テクニックです。

では、ごゆっくりと秘伝の世界を楽しんでください……。

二の章　秘伝の問答八六

1 セグメントの概念

バブバブー。バブちゃんは3歳ですぅ。でも、もう英語と日本語とフランス語とドイツ語がペラペラですぅ……バブ。ここまでくると、どんな国の言葉でもすぐ覚えられるようになるのですぅヨ。生まれた時から、そういう環境に住んでいたから、言葉に対する違和感なんてないんですぅ……バブ。
もう人間の言葉は飽きたから、次にマシン語を覚えたんですぅ……バブ。それは、8ビットの代表格Z 80ですぅ。でも、あれは思ったより簡単だったから今度は16ビットの代表格8086に挑戦しようとしているのですぅ……バブ。バブちゃんの覚える方法は、わからないことがあったらすぐ聞いて覚えてしまうことですぅ。こういう時は、小ちゃいほうが得ですねェ……バブゥ。
質問というのは、セグメントという実体がよくわかんないんですぅ。バンク切り換えでもなさそうだし、いったいこれはどういったものなんですぅ……バブ？　バブちゃんにもわかるように、やさしく教えてちょうだいですぅ。では、バブバブゥ。

芭蕪夫・輪田智（スイス）

答

スイスという国は、北海道の半分ほどの狭い国土の中にドイツ語、フランス語、イタリア語、ロマンシュ語が公用語として存在しており、しかもほとんどの人が英語を話せるというから驚いてしまいます。

バブちゃんの才能は、きっとそんな環境の中で生まれ育ったのでしょう。それにしても驚きですぅ……バブ。

実は、Z 80などの8ビットCPUから8086系の16ビットCPUに移った場合、このセグメントの概念がひとつの壁になっているケースは少なくありません。ベルリンの壁も崩壊したことですし、ぜひセグメントの壁も取り崩してしまいましょう。まず、8086系CPUの大きな特徴を改めて確認してください。

①レジスタは16ビットで構成される
②メモリのアクセス範囲は1Mバイトである

これら2つの特徴から、必然的にセグメントの概念が生じてきたのです。16

ビットで数えられる範囲は、ご存じのように0～64Kバイトです。ところが、メモリのアクセス範囲は1M（64K×16＝1024K）バイトですから、16ビットでは全メモリ空間を示すことができません。そこで、特別な工夫をすることにしたのです。

この特別な工夫（セグメントの概念）とは、簡単に言うと次の問に対する答のようなものです。

10本の指で100まで数えてください

これに対する代表的な答は、左手の指に10の重みを持たせ、右手の指で10数える毎に左手の指を1本づつ折っていくという方法でしょう。セグメントの概念は、まさにこのことなのです。2つの16ビットの数値の一方の刻みに、16の重みを持たせることにより1Mバイトを表現するのです。

この16の重みを持たせた数値のほうをセグメントアドレスと呼びます。1つの刻みが16の重みを持っていますから、セグメントアドレスが1増えることはメモリ上で16バイト増えることを意味しています。そして、もう一方の16ビットの数値は、オフセットアドレスといい、このセグメントアドレスにオフセットアドレスを与えることによりメモリ位置を決定するのです。すなわち、実行アドレスの計算は、セグメントアドレスを16倍した数値にオフセットアドレスを足すことによって求められます。また、オフセットアドレスで数える0～64Kの範囲を1つのセグメントといい、セグメントアドレスがこの1つのセグメントの開始アドレスを決定しているのです。

8086は4つのセグメント用レジスタによって、全メモリ空間を管理しています。

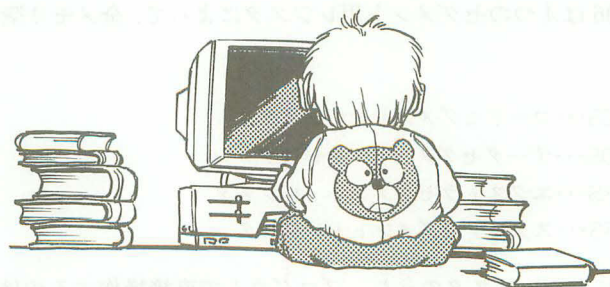
CS･･･コードセグメント・レジスタ
DS･･･データセグメント・レジスタ
ES･･･エクストラセグメント・レジスタ
SS･･･スタックセグメント・レジスタ

これらのレジスタのうち、プログラムで直接操作するのはDSとESです。特

に、DSに対しては暗黙に指定されている命令が豊富にありますから、頻繁にアクセスするデータはデータセグメントに設定するようにします。一方、ESレジスタに対する命令は、わずかにストリング命令があるだけで、あとはセグメント・オーバーライド・プリフィックス命令を使うことになりますから、同時に複数のセグメントをアクセスする場合に用いることが多いようです。

セグメントに慣れないうちは、プログラムの先頭でDS=ES=CSとし、プログラムでBPレジスタによるメモリ参照を行わなければ、1つのセグメント内だけでプログラム作りができます。こうすれば、セグメントをまったく意識せずに済みますから、初心者でもプログラミングそのものに集中できるようになります。

いずれにしても、いつかはセグメントを理解した上でプログラミングしなければなりません。バブちゃんも、すぐにそういう日が来ることでしょう。セグメントでも何でも、すべては慣れなのですゥ……バブ。

2 マイナスの数値

はじめまして。ボクはこれまでゲームばかりやっていたので、学校では「動くパソコンソフト」と言われています。でも、実は秘かにゲームから脱皮しようと思い、マシン語を勉強し始めています。マシン語といっても、やっとニモニックの意味とアセンブラの必要性がわかりかけてきた程度です。

ところでボクの読んだ本によると、一般にアセンブラのソースリスト上では十進数表記も可能と書いてあります。といっても、ニモニックで用意された命令で扱える数値は1バイト（0～FF_H）か2バイト（0～$FFFF_H$）ですから、十進数でいえば0～255まで（1バイトの場合）か、0～65535まで（2バイトの場合）となるはずです。でも、でもでも、ですよ。ある時、雑誌に……

```
XPOS DB  -10
```

となっているソースリストがあったのです。ハッキリ言って、これがどういう数値を表すのか意味がわかりません。もちろん、十進数ならわかります。いったい、十六進数でマイナス(−)とはいくつのことですか。ちなみに、ボクはまだ本物のアセンブラは持っていません。早くアセンブラがほしい……。

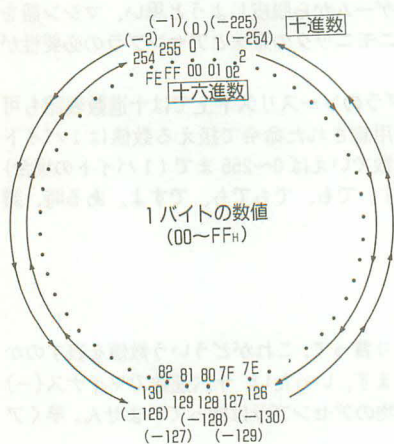
初心者マー君（埼玉）

答

初心者からの質問というのは未知数の魅力にあふれていて、内容にかかわらずとても新鮮に感じます。ついつい自分の過去を思いだしたりします。

マシン語が上達するひとつのカギは、自分の持っているアセンブラの能力をどこまで発揮できるかにあります。とはいえ、最初から高級なアセンブラを持てばいいというものでもありません。あまりアセンブラが高級だと、マシン語を覚える前にアセンブラのマニュアルに圧倒されてしまうからです。だから、最初は操作が簡単で安価な入門用アセンブラを手に入れ、自分の技術レベルに合わせてグレードアップするほうがいいでしょう。

ということで、初心者マー君の疑問に一気に答えてしまいましょう。なーに、答は簡単です。マシン語で扱う数値は、使う人の気分次第でプラスにでもマイ

ナスにでもなるのですから。つまり、ループ(FF_Hの次は0)している数値を前向き(プラス方向)に数えるか、後ろ向き(マイナス方向)に数えるかの違いがあるだけなのです。

だから、−10は十六進数でいえば$F6_H$というわけです。もし、ここでの命令が「DB」でなく「DW」であれば、−10は$FFF6_H$となります。でも、でもでも、ですよ。これで若葉マークが取れたなんて思ってはいけません。なぜなら、演算の結果、サインフラグを見て条件分岐をさせる場合は、コンピュータはデータの最上位ビットを見て+/−を判定するからです。つまり、数値としてはすべてをプラスと考えようがマイナスと考えようが自由ですが、コンピュータにその判断をさせる場合は、次のようにキチンと区別をしなければならないということです。

1バイトの場合:	00_H〜$7F_H$	= 0〜127	(プラスの数値)
	FF_H〜80_H	=−1〜−128	(マイナスの数値)
2バイトの場合:	0000_H〜$7FFF_H$	= 0〜32767	(プラスの数値)
	$FFFF_H$〜8000_H	=−1〜−32768	(マイナスの数値)

マイナスという感覚で数値を扱う時は、ここまで理解した上で使用しないと思わぬバグに陥ることがあります。そして、こういう一見つまらなさそうな理由によるバグが、実は最も恐ろしい落し穴なのです。しかし、マイナスが自由に使えるようになると、プログラムが急に進歩するのも事実です。初心者マー君がプログラ・マー君になる日も近いことでしょう。

なお、雑誌などに発表される簡易的なアセンブラでは、マイナスが使用できないものもあります。プログラム技術の向上のためには、そのようなアセンブラからはできるだけ早く卒業し、互換性の高いMS-DOS上のMASMを準備したいところです。

3 慣用句的論理演算

ボクはマシン語を覚え始めたばかりの小6です。まだ本当に短いプログラムしか組めません。ボクのマシン語の先生は中2の兄ですが、その兄もRPGでいうならレベル5くらいみたいです。ボクはレベル2くらいでしょう。
でも、兄は論理演算について一応知っています。ボクも兄に教わったので、論理演算をするとビットがセットされたり、リセットされたりするということはわかるようになりました。ただ、まだそれがどんな時に必要になるのかはわかりませんけど。
ボクは、いま昔の簡単なゲームを見つけて、それを逆アセンブルしてプログラムの仕組みを覚えようとしています。昔のゲームっていうのは、これが本当に商品だったのかと思うほどレベルが低かったのですね。プロテクトなんて、もちろんナシです。でも、結構論理演算なんかも使っているみたいです。
ところが、よく出てくる論理演算に「XOR　AX,AX」と「OR　AX,AX」というのがあります。紙に書いて結果を確認すれば、AXレジスタの値がいくつになるかはわかります。でも、なんのためにやっているのか、その目的がさっぱりわかりません。昔のゲームといっても、結構むずかしいんですね。どうかよろしくお願いします。

少年アセンブラ（岐阜）

答

こりゃマイッタ!?

小学6年生でマシン語を始めてる……だって!?　それも、論理演算にまでコマを進めてる……だって!?　オドロキ桃の木サンショの木……です!!

しかも、マシン語の勉強法に昔のゲームソフトを解析するなんていうのは、将棋でいうなら最初から最善手を指しているようなベストウェイです。なにしろ、昔のゲームというのはプログラムの内容がほとんど画面に反映されるようになっていますから、解析してプログラム手順を覚えるには最高の教科書なのです。もしかすると、アッという間に先生であるお兄さんのレベルを越えてしまうかもしれません。

……が、そうなるためには今回の質問のようにプログラムの目的をしっかりと理解することが大切です。だから、この本が手元にあるってことは、もはや虎の巻を手に入れたようなものです。

さて、まず論理演算による各ビットの変化についてですが、これはマシン語の入門書であればどんな本にでも載っていることなので、ここでは省きます。問題は、いかにして論理演算の用途を把握するかということです。

その理由は、論理演算については純粋にビット操作が目的の場合と、フラグを変化させるのが目的の場合があるからです。ビット操作が目的の場合は、そのビットの役割を調べればわかりますが、フラグを変化させることが目的の場合はいくらビットの内容を調べても結論は出てきません。

では、どうすればいいかというと、単純に暗記すればいいのです。おもに使用される慣用句的論理演算は次の3種類ですから、暗記というほど大袈裟なことでもありません。とはいえ、使用頻度はビット操作が目的の論理演算より多いくらいですから、とりあえずは深く考えずに覚えてしまいましょう。

XOR　reg,reg

演算の結果が常にゼロになりますから、「MOV　reg,0」の代わりによく用いられます。

ただし、演算によるフラグ変化がありますから、そのことを考慮した上で使い分けなければなりません。

OR　reg,reg

論理演算によるフラグ変化を利用し、「CMP　reg,0」の代用として、あるいはキャリーフラグのリセット用としても使われます。

AND　reg,reg

「OR　reg,reg」とまったく同様の目的で使用されます。

(注) 第1オペランド(reg)と第2オペランド(reg)は同じレジスタ。

これらは単なる慣用句ですから、論理演算の機能を本当に利用したとはいえないのですが、プログラムがちょっぴりオシャレしたような気分になれるでしょう。しかし、本当にカッコイイのは論理演算を2つ以上組み合わせて目的のデータを作り出す、というような使い方をしたプログラムなのです(例えば、XORをとってからANDをとる……等)。

その時には、マシン語のレベルも10を超えているでしょう。でも、小6でそこまでカッコイイと、ハッキリいって大人はつらい……。

4 レジスタの0チェック

私は現在イギリスはブライトンに住んでいます。本当の目的は一人旅ですが、最初の8週間は英会話のスクールに通い、まずは外国生活に慣れるつもりです。いま、ちょうど半分が過ぎたところですが、夏なのにカラッとした気候のブライトンにいると、このまま住み着きたくなるほどです。

でも、日本では大学5年生に籍を置いたままですから、スクールの後は予定通り旅をして日本へ帰ります。帰国予定は12月です。卒業のほうは、具体的には秘密ですがちゃんと手を打ってありますから大丈夫でしょう。

ところで、どこにいても気になるのは覚えかけたマシン語のことです。だから、英語の辞書とマシン語の本はいつもカバンに入っています。その本には……、

『レジスタがゼロかどうかを判定するには、「CMP　AX,0」より「OR　AX,AX」がいい』

と、書いてあります。理由はプログラムに要するメモリが少なくて済むからだそうですが、他にもレジスタのゼロチェックの方法があると聞いたことがあるのですが、何かうまい方法があったら教えてください。

W大商学部の星（ブライトン）

答

ム、この男性はどこかで見かけたことがある……。それも、つい最近などというものではなく、なぜか生まれた時から現在まで、そのすべてを知っているような気がしてならないのだが……それが誰なのかはわからない……ウ〜ム。

マシン語を始めたばかりは、プログラムの長さとか速度などまるで気にならないものですが、やがてそれに目覚める時がくるのです。その時こそ、マシン語が新たなるパズルゲームに発展する日なのです。あなたにも、ついにその日がやって来ましたか。

質問の回答をする前に、なぜ「CMP　AX,0」より「OR　AX,AX」がいいのか考えてみましょう。答は簡単です。「CMP　AX,0」では、命令にメモリを3バイト使用し、実行に要する時間は4クロックかかりますが、「OR　AX,AX」ならばメモリは2バイト、実行時間は3クロックで済むからです。もちろ

ん、これらの命令はAXレジスタに限らず使うことができます。ただし、すべてのフラグが同じ変化をするわけではありませんから、ソックリ同じとはいえませんが……。そのことを知った上でなら、実用上は同じと見なしても差し支えないでしょう。

早い＝タイムクロック数が少ない

安い＝使用メモリ数が少ない

これが、うまいプログラムというわけです。では、質問への回答です。結論からいうと、演算の結果に従ってゼロ・フラグは変化するが、レジスタの値は変化しないという条件を満たせばよいことがわかります。しかも、「CMP reg, 0」より、使用メモリまたはクロック数で有利でなければなりません。そこで、こういう場合には次のようにします。

```
DEC    reg16
INC    reg16
```

または

```
INC    reg16
DEC    reg16
```

どちらも命令に要するバイト数、実行クロック数は同じです。足して引いても、引いて足しても、変化するのはフラグだけですから、レジスタの内容は壊れないわけです。まさに、パズルではないですか……。

ここでレジスタを16ビットに限定していることに注意してください。もし、使用レジスタが半分の8ビットレジスタの場合には、逆に「CMP」命令を素直に使ったほうが有利になります。なお、1つの命令に対するバイト数と、クロック数は三の章の8086ニモニック表を見ればわかります。クロックというのは、その命令に必要な時間の単位ですが、実際の時間はそのマシンの基本周波数（8 MHzとか10 MHz）によって変わります。

5 「LEA」の効用

私は鉄道マニアです。JRの全路線を踏破するのが目標ですが、お金と時間のかかることなのでいつ達成できるか、自分にもさっぱりわかりません。
そこで、パソコンを使ってひと足お先に画面上で全国踏破しようと思うのですが、今度はプログラムで頭を使いデータ入力で時間を使い、やっぱりいつ達成できるのかわかりません。それでも、こちらのほうはお金がかからないだけ助かってはいますけど……。
実は、最初はプログラムを BASICで組んでいたのですが、あまりに遅いためマシン語でやろうと決心したのです。ただ、マシン語でまともなプログラムを組むのはこれが初めてなので、マシン語といっても幼稚なものです。そこで、恥ずかしいのですが未熟者からの質問と思って教えてください。よく雑誌などで……

> [LEA ****]命令よりも[MOV OFFSET ****]命令のほうが実行スピードが早いので、LEAの代わりにMOV命令を使用しています。

……と、書いてある投稿記事があります。確かにニモニック表ではLEA命令は2+EAクロックで、MOV命令は4クロックです。これではLEAの存在する意味はナイも同然です。これは一体どういうことでしょう。
LEAという命令が存在している以上、なんらかの意味があると思うのですが、ちなみに、現在のプログラムは乗車ルートを決めるという段階で、一応思考ルーチンとなっています。本当に意味がないのであれば、思考時間の短縮のためにLEAをMOV命令に代えるべきでしょうか?

LEA不信のJRマニア(神奈川)

答

この答は簡単ですね。LEA命令だって人の子、いやマシンの子です。存在するからにはそれなりの意味があるのです。まずは質問にあるような単純な場合を考えてみましょう。

```
        ⋮
MOV    SI,OFFSET DATA1
        ⋮
```

4クロック

```
        ⋮
LEA    SI,DATA1
        ⋮
```

8クロック

このように、ダイレクトメモリアクセスの場合には、LEA命令は2+EA(EA=6)でトータル8クロックとなり、MOV命令の倍の時間がかかることになります。

そこで、まずはこのクロックというのは具体的にどの程度の時間をいうのか計算してみましょう。ただし、割り込みやDMAなどによるウェイトは考慮に入れません。また、基本周波数は10 MHzとしています。

1クロック=1÷10÷1000÷1000 秒
　　　　=0.0000001秒

これが1クロックの実際の長さです。つまり、LEA命令をMOV命令に置き換えると、それだけで 0.0000004秒高速になるというわけです。

しかし、だからといって、LEA命令が不要だというわけではありません。LEA命令は、インデックスレジスタ等によるメモリ参照に対するオフセットアドレスを得たい時にも活用できるのです。例えば次のような場合です。

```
      ⋮
LEA   SI,[BX+DI+DISP 1]
      ⋮
```

これで、SIレジスタへオフセットアドレスが格納されます。もちろん、このSIに求めたオフセットアドレスを単純に使う場合であれば、なにもSIへオフセットアドレスを求めるまでもなく、このアドレッシングをそのまま使えばすむことですが。求めたオフセットアドレスをさらに加工したり、データエリアのアクセス範囲をチェックしたり、それなりに価値がある命令なのです。決して存在価値がナイだなんて思わないでください。

なかには、「MOV　reg,OFFSET ＊＊＊＊」よりも意味がハッキリとするという観点から、積極的に使っているLEA信者もいるくらいです。こうなると、どちらを使うほうがよいかは、趣味の問題となってしまいます。まあ、質問者の例では、プログラムの高速性からMOV命令を使っても問題はないでしょう。

6 RAMの特徴を活かす

ドスコ〜イ!!
朝もハヨからドスコ〜イ。ワシは大横綱千代の富士。
……を目指して、新弟子検査を受けようとしている自称超横綱万代の富士(まよのふじ)っす。身長183センチを目標に、168センチの身体にムチ打って連日ドカ弁を食べているっす。おかげで体重だけは115 kgもあるっす。ア、いま中3っす。
寝てもさめても相撲だったのに、なんの間違いか誕生祝いにパソコンがほしいと言ってしまったっす。もちろん冗談100パーセントのつもりだったす。これまでほしいものを一回で買ってもろたことなんてなかったすから。ところがなんと、誕生日の日学校から帰ると、机の上には本物のパソコンが……。
てなわけで、最近ではBASICを通り越してマシン語に夢中っす。ところで、メモリにはROMとRAMがあるっすが、ROMだけじゃプログラムは無理ってことは理解してまっす。ワークエリアがなけりゃプログラムは組めないっすもんね。
でも、この前『RAMの特徴を利用したプログラム』があるってことをどこかで耳にしましたっす。これはいったい、なんのこっすか?

どすこいドカ弁（香川）

答

いよいよマシン語も体力勝負の時代に突入したようですね。ドスコ〜イのかけ声と共にマシン語が土俵の外へ突き飛ばされそうです。

さて、プログラムであればBASICであれマシン語であれRAMは絶対に必要ですが、これは基本的には変数保持のためにワークエリアが必要だからです。例えば、カーソルやキャラクタの表示位置を示すワークエリアとして、よく「XPOS」「YPOS」というラベル名を使用しますが、この中身をプログラム中にBXレジスタにロード（BLレジスタにX座標、BHレジスタにY座標）すると仮定してみましょう。

これを通常のプログラム－①とRAMを利用したプログラム－②とに分けて組んでみたのが次の2つです。どこがどう違うか、それぞれの特徴を見極めるつもりでプログラムを見てください。なお、プログラムの右側にある数字は、その命令に必要なバイト数を表しています。

```
① XPOS  DB   0                        1
   YPOS  DB   0                        1

   WPOS  PROC
               ⋮
         MOV  BX,WORD PTR CS : XPOS    5
               ⋮
               ⋮
               ⋮
   WPOS  ENDP
```

```
② MOV_BX EQU   0BBH


   WPOS  PROC
               ⋮
         DB    MOV_BX; ┐              1
   XPOS  DB    0     ; │ MOV BX,0     1
   YPOS  DB    0     ; ┘              1
               ⋮
   WPOS  ENDP
```

ここで②の先頭データ0 BB_Hが「MOV BX,･･･」のオペレーション・コードであることに注意してください。また②は、見かけ上データの集まりのようにも見えますが、逆アセンブルしてみると「MOV BX,0」をハンド・アセンブルしたものとなっていることがわかります。

どちらのプログラムも、これ以外の場所で「XPOS」「YPOS」の中身を操作する命令に関してはまったく同じ条件です。ということは、RAMを利用したプログラムー②では、通常のプログラムー①より命令に要するメモリ数が4バイト少なくて済むということがわかります。

さらに、BXレジスタにデータをロードする命令も、①(16クロック)に対し②は4クロックと1/4になっています。したがって、この3行分だけに関していえば、メモリ数もタイムクロック数も②は①をかなり節約した形になっているわけです。こうなると、どちらが得かはもう明白です。いかにも、①のプログラムは素人っぽく見えてきますね。

とはいえ、これはテクニックとしてはかなり特殊な部類に属し、かつ思わぬバグに陥る危険性も秘めているのです。不用意に利用するのではなく、メモリの節約と速度の追求がどうしても必要な時だけに限定しなければなりません。というのは、8086には実行速度の高速化を図るため、6バイト先読みの命令キュー(Instruction queue)があるからです。この特徴を十分に考慮しないと、迷宮入りのバグに泣かされるかもしれません。簡単な例として、次のようなプログラムを考えてみましょう。

```
MOV_BX EQU     0BBH

                 ⋮
         MOV     AL,12H
TST1:    MOV     CS:XPOS,AL
         DB      MOV_BX;─┐
XPOS     DB      0     ; │ MOV BX,0
YPOS     DB      0     ;─┘
                 ⋮
```

この例では、TST1というラベルの命令が実行される時、すでに「MOV BX, 0」もCPUの命令キューに格納されています。したがって、プログラム実行後のBXレジスタの値は依然0のままです。もちろん、XPOSのメモリの値を確認してみると12_Hとなっていますから、この先読みに気が付かないと永遠に悩むことになってしまうのです。

また、このようなプログラムが通用するのはプログラム自体がRAMに置かれるということが絶対条件ですから、将来ROM化するような場合はキチンとワークエリアを分離した①の書式にしなければなりません。あくまでも「RAMの特徴を利用した」ということを忘れてはならないのです。さらに、他の言語系へ移植をする可能性がある場合や、多人数でプログラム開発をするような場合には、十分検討してからにしてください。

しかし、本書のプログラムにおいては、原則的にROM化することまでは考えていませんから、この先さらにRAMの特徴を利用したプログラムが現れてきます。その際、特にRAM専用という断りを入れませんので、もしROM化予定のプログラムに本書のテクニックを応用したい場合は、それなりにRAMエリアを活用するよう、各自で考慮してください。

そもそも、タダで便利なものにはどこかに条件があるというのは世の常なのです。香川のどすこいドカ弁君にパソコンがプレゼントされたのも、きっとその裏には別の願いがあったんではないっすか。身長／体重のバランスからして……。

7 FARタイプのプロシージャ

この顔、どう歌手の森田進一に似ているでしょ。次は岩崎ひとみ、その次は三木ひろしと村田春夫の合成顔……。
おっと、手紙では自慢の百面相がお見せできなくて残念無念。私は、芸能界一のモノマネ男と言われているメンチで〜す。気楽な商売に見えるでしょうけど、これでも結構大変なんですヨ〜ッ。常に新ネタを考えないと忘れられてしまいますからネ。
そこで、芸能人の顔の特徴と自分でできる顔の特徴をコンピュータに登録し、ネタに困ったら「次は野口四郎のマネが可能です」というような回答が出るように、秘かにマシン語でプログラムを組もうと思っているのです。
一応、セグメントについては勉強して理解していますが、どうも異なったセグメント間でのサブルーチンの定義やコールの方法がわかりません。なにしろ、プログラミングはモノマネ芸と違って素人ですから……。
どうか、みんなに内緒でソッと解説してください。これも新ネタと同じで、いわゆる企業秘密としたいので〜ス。

メンチ（芸能界）

答

テレビで華々しく活躍中のメンチさんも、やはり陰では人知れず苦労をしていたのですねェ。しかも、モノマネ同様、イヤそれ以上にプログラムに対するテーマが素晴らしいではないですか。プログラムの勉強をしても、いざプログラミングとなると「何をプログラムしていいかわからない」という人が多いのですが、このあたりはさすがプロの芸人といえるセンスのよさで〜ス。

「……で〜ス」だけマネしてもモノマネになりそうもないので、素直に質問に答えることにします。本書では、原則としてMS-DOS専用のアセンブラ「MASM」の使用を前提としていますが、MASMでは１つの独立した処理体系をサブルーチンとはいわずにプロシージャと呼んでいます。

異なるセグメントにおけるプロシージャは、単にプロシージャのタイプをFARとするだけで、FARタイプのサブルーチン（プロシージャ）が定義されます。もちろんRET命令も自動的にFARタイプに変換されます。実際に、セグメ

ントCODE 0からCODE 1に定義したFARタイプのプロシージャTEST 1をコールする例を次に示します。

```
CODE 0  SEGMENT
          ⋮
        CALL   FAR PTR TEST 1
          ⋮
CODE 0  ENDS

CODE 1  SEGMENT

TEST 1  PROC   FAR   ┐
          ⋮          │
        RET          │ FARタイプのプロシージャの定義
TEST 1  ENDP         ┘

CODE 1  ENDS
```

なお、定義したFARタイプのプロシージャに複数のRET命令を使った場合でも、それらはすべてFARタイプに変換されます。しかし、プロシージャを定義する場合は、プログラムの保守性やわかりやすさなどの点から、できるだけ入口と出口は1つにするよう心がけるべきです。

また、割り込みプロシージャの場合には、割り込み専用のIRET命令が使われます。これは、割り込み処理へ制御が移動した場合、CS、IPと共にフラグがPUSHされるため、最後のIRET命令ですべてをPOPするためです。

今度は、こちらからお願いです。ぜひ、百面相モノマネの秘伝を公開してください。待っていま〜ス。

8 無駄命令とは

青春とは爆発だァ!!

オレは高3。あり余るエネルギーを、走ることで燃焼させている。別に陸上部に所属しているわけじゃないから、距離はたったの3kmさ。だけど、ほとんど毎日のように走っている。正直いって走っている時はメチャ苦しい。顔は苦痛にゆがみ心臓はオーバーヒート寸前……。ここで歩けばどんなに楽かと、いつもそれだけを考えて走っている。

でも、それを我慢して走り終わると、これがまた何とも言えない壮快感。誰もホメてくれなくても、オレはその瞬間が大好きなんだ。無駄なことをしている、なんて言うヤツもいるけどネ。無駄なことができるってのは、ゆとりがある証拠さ……。

ところで、コンピュータにも無駄命令ってのがあるって話だけど、こればかりは自分のことじゃないからよくわからない。ニモニック表を見たら「NOP」という命令があったけど、これが無駄命令のことなんだろうか。

まさか、パソコンにもエネルギーのあり余る時があるっていうんじゃないだろうな。よくわからないから、ひとっ走りしてこようっと!!

人間無駄蒸気(秋田)

答

いいなァ、無駄なことができるなんて!!

……と言いつつ、私もまだ現役で走っています。でも、それはエネルギーが余っているからではなく健康のためです。もちろんエネルギーは消耗しますが、これは回復力という別のエネルギーを呼び出してくれますからね。決してエネルギーの無駄な消費ではないわけです。

つまり、一見して無駄に見えることが実は無駄なんかではなく、身体にとって重要な役割を果しているということです。コンピュータにおける無駄命令も、本当に無駄なら誰も使わないはずです。と考えると、無駄命令という言葉が存在すること自体、そこには無駄ではない何かがあるはずです。

そこで、まずは質問にあった「NOP」という何の役にも立ちそうもない命令を見てみましょう。確かに、この命令はフラグ変化も伴わないし、実行結果を見ても何も残してくれません。でも、これは決して不要な命令なんかではない

のです。その証拠に、次のような役に立つことを実行してくれています。

(1) メモリを1バイト確保する
(2) 3クロックという時間を消費する

メモリを1バイト確保してくれるということは、必要に応じてそのメモリを利用できるということです。例えば、衝突をチェックするサブルーチンがあるとすると、その先頭に「RET（=C 3_H）」を入れれば、簡単に不死身モードができてしまいます。

```
CHECK: NOP
         ⋮
       RET
```

```
CHECK: RET
         ⋮
       RET
```

また、3クロックという時間の消費は、外部機器や特殊デバイスとのヤリトリをする場合に、タイミングを取るための最小ウェイトとして活用されます。

どうですか。これら2つの役割だけでも「NOP」が役に立つ命令であることがわかりますね。

さて、無駄命令といわれているものには、このほかにもタイマーとしてのウェイトルーチンがあります。例えば、カーソルなどはウェイトを入れなければ早すぎて思い通りに動かせませんし、ゲームでもスピードの調節が不可欠です。このような場合、正確には割り込みによってキチンとしたウェイトを取らなければなりませんが、簡単なルーチンの場合は次のようにループによってウェイト調整をすることもあります。

```
WAIT    PROC
        MOV   CX,30     ←CXの値によって長さを調節する
WAIT1:  PUSH  AX
        POP   AX
        LOOP  WAIT1
        RET
WAIT    ENDP
```

このようなプログラムを一般に無駄命令と称しているわけですが、本当に無駄なのはダラダラと不要に長い下手なプログラムというべきです。必要があるから使われているウェイトルーチンは、かわいそうにも俗称だけが無駄命令というわけです。

もっとも、人間の反応がコンピュータ並みに早ければ、大半の俗称無駄命令も省けることになりますが……。そうなると、コンピュータそのものが無駄な存在になってしまいます。

9 ESレジスタの利用

私は、定年退職してからパソコンを始めた者です。息子のパソコンをいじっているうちに、ついマシン語なるものに興味をもちました。ところが、年のせいか、どうも融通がききません。いつも、レジスタが足りなくて「PUSH/POP」ばかりを使ってしまいます。もちろんワークエリアの存在は知っています。しかし、これはワークエリアを使っても解決するような問題ではないと思うのです。
いま、DSレジスタに800_Hを足したいのですが、あいにくESレジスタしか空いていません。なんとか、「PUSH/POP」地獄から抜けたいのですが……。

```
PUSH   AX
MOV    AX,DS
ADD    AX,800H
MOV    DS,AX
POP    AX
```

やっぱり、これしかナイものでしょうか。私にはこれ以上の案はでませんが、老人でもプログラムの格好は気になるものです。

ボケない老人(栃木)

答

パソコンに年齢なんか関係ありません、という見本のような例です。コンピュータといえば、ついこの間までは大型の電子計算機を意味していたのですから、若い人も年とった人もスタートは一緒です。

そんなことより、新しいことにチャレンジできるなんて、まさに青春まっ盛りではないですか。おまけに、「PUSH/POP」地獄から脱却したいとは、なんて素晴らしいことなのでしょう。プログラムの格好を気にするなんていうのは、プロのプログラマーでもなかなか言えないセリフです。

ところで、このようなケースには実際によく出会います。特に、データの転送などではDS:SI/ES:DIレジスタでデータのアドレスを指定し、CXレジスタをカウンタに使うことが多いですから、途端にレジスタ不足になってしまいます。

「PUSH/POP」命令は確かに便利ですが、ESレジスタが空いているならば、これを利用しないのは、やはり無駄といえるでしょう。もちろん、ESレジスタには演算命令がありませんからワーク・レジスタとして活躍してもらうことになります。次の例を参照してください。

```
MOV   ES,AX
MOV   AX,DS
ADD   AX,800H
MOV   DS,AX
MOV   AX,ES
```

質問にあった「PUSH/POP」を使用した方法では、使用メモリ数＝9バイト、クロック数＝26となります。一方、こちらの方法では、使用メモリ数＝11バイト、クロック数＝12です。たったこれだけでも、14クロックも節約したことになります。もし、ループ内処理であれば、14×N(ループ数)クロックの節約ですからかなりのものです。セグメント・レジスタといえども遊ばせておくことはないのです。

では、栃木のおじいちゃん、格好のいいマシン語プログラムを目指して、ますます頑張ってください。

10 「XCHG ES,DX」を実現

コンニチワ。こちらは冬が長い北の国です。寒い日は、家の中でパソコンをするのが最高です。特に、マシン語をプログラムしていると時のたつのを忘れます。今はまだ入門者ですが、そのうちアッと驚くプログラムを組んでみせます。
ところで、いつも不便に思うことがあります。それは、レジスタのXCHG命令についてです。汎用レジスタには「XCHG　reg,reg」命令があるのに、セグメント・レジスタにはありません。だいたい、セグメント・レジスタには演算命令がありません。どう考えてもこれは不便なレジスタだと思いませんか？
今は次のようにしています。もっとよい方法はあるでしょうか。

```
PUSH    AX
MOV     AX,ES
XCHG    AX,DX
MOV     ES,AX
POP     AX
```

夜ふかしヒグマ（北海道）

答

雪がしんしんと降る中をプログラムか……。うらやましい環境だなァ。きっと、プログラムに疲れたら、近くの山にスキーにでも行くんでしょうね。いいプログラムはいい環境のもとで生まれるとよくいいます。タバコの煙と、ほこりにまみれた部屋からは、それなりのプログラムしかできないという気がします。

きっと、近いうちに日本を揺るがすようなプログラムができることでしょう。なんといっても、本人がその気になっているのが強みです。

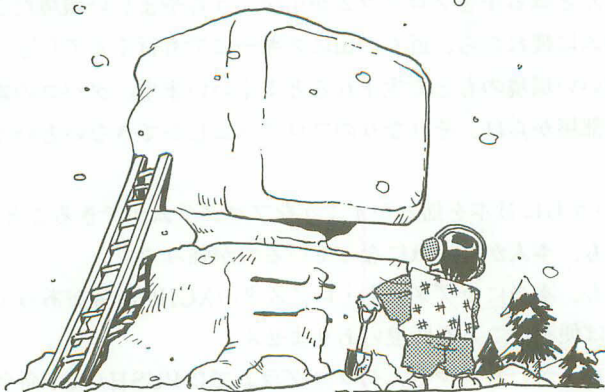
それにしても、本当にセグメント・レジスタのXCHG命令があってもよさそうです。あれば便利なことも間違いありません。

ところで、夜ふかしヒグマさんのプログラムではPUSH/POP命令を使っていますが、これをうまく使うともっと簡単になります。いわば、コロンブスの

卵のようなものです。

```
PUSH    ES
MOV     ES,DX
POP     DX
```

この例のように、PUSH/POP命令は同じレジスタでなければならないという制限はありませんから、意図的に使う場合もよくあります。また、これを利用するとセグメント・レジスタの初期化も汎用レジスタを介さずにできます。

```
MOV   AX,CS          →      PUSH   CS
MOV   DS,AX                 POP    DS
```

11 セグメント・レジスタの初期化

ここは魔法の国です。といっても、どこにあるのか知らないかもしれません。でも、本当にあることだけは信じてくれると思います。だって、ボクの姉さんの友だちは地上に降りて正義のために活躍しているそうですから……。
ここでもパソコンは盛んですが、プログラム言語はマシン語以外は使用禁止となっています。なぜかというと、マシン語程度ができないようでは、とても魔法を使いこなすことなんてできないからです。それに、ここだけの話ですけど、マシン語は魔法の呪文の練習にもなるそうです。ボクは、ニモニックを覚えたばかりの初心者ですから詳しくはわかりませんが、とにかくマシン語をマスターしなければならないのです。
そこで質問です。ある時「セグメントの初期化にはPUSH/POP命令を使うといい」と知ったのですが、これしか方法がないのでしょうか？　というのは、なんとなく妙な感じがするからです。あくまでも、これはボクの感ですけど……。では、初心者のボクにわかるように教えてください。
ア、地上に降りた姉さんの友だちの名前はサリーっていいます。

魔法使い見習いジロー（魔法の国）

答

魔法の国……。サリーちゃん……？　どこかで耳にしたことはありますが、まさか実在するとは思いませんでした。もちろん、マシン語が魔法の練習になるということも初耳です。なんだか、私にも魔法が覚えられそうな気がしてきましたが……。きっと地上では無理なんでしょうね。

さて、セグメントの初期化とは問10に示した内容だと思いますが、実はこの方法はどちらかというと特殊な例なのです。というのは、PUSH/POP命令はクロック数がかなりかかる命令ですから、空いているレジスタがある場合には、そのレジスタを介して間接的に初期化したほうが実行速度が速いのです。まず、それぞれのクロック数を比較をしてみましょう。

※PUSH/POPの場合

```
PUSH   CS      10
POP    DS    + 8
             ────
             18クロック
```

※レジスタを使った場合

```
MOV   AX,CS     2
MOV   DS,AX   + 2
              ────
              4クロック
```

ご覧の通り、トータルで14クロック違いますから、問10の初期化方法は空いているレジスタがない場合に有効ということになります。セグメント・レジスタを初期化する方法としては、他にも次のような方法があります。

①メモリからロードする

```
DSREG   DW    SEGMENT 1
          ⋮
        MOV   DS,CS:DSREG
          ⋮
```

②他のレジスタとペアでロードする

DSレジスタと16ビットレジスタをペアでロードするLDS命令と、ESレジスタと16ビットレジスタをペアとするLES命令がある。

```
REGDT   DD    セグメント値：オフセット値
          ⋮
        LDS   SI,CS:REGDT
          ⋮
```

では、早くマシン語をマスターして一流の魔法使いになってください。そして、サリーちゃんみたいに地上に降りて、今度は私に魔法を教えてくださいナ。

12 「EXX」命令を実現

与論島をご存じですか。そう、若者の島、ヨロン島です。沖縄が日本に返還されるまでは、日本最南端の島といわれていました。
わたしの家は民宿をしています。ヨロンにはヨロン憲法（献奉）という歓迎の挨拶があって、島に来た人はまず焼酎をなみなみと飲まされるんですよ。わたしの役目は飲むふりをして飲ませることです。だって、次の日は学校だもん。
ここの海岸にはコンペイ糖みたいな形をした星の砂がたくさんあります。有名だから知ってますよね。わたしは、いま「星の砂」というタイトルのZ80用のゲームを8086用に組み直しています。
でも、Z80にあって8086にはない命令があって困っています。それは裏レジスタへチェンジするEXX命令です。こういう場合、8086ではどのようにすればよいのでししょうか。

星降る乙女（与論島）

答

ヨロン島の星降る乙女ちゃんの作品で「星の砂」……。もう目の前にメルヘンの世界が広がってきそうです。ヨロンには潮が引くと沖合いに島のように現れる百合が浜なんていうのもあるし、夢がいっぱいですね。

でも、お酒を飲めない人にとっては、ヨロン憲法は余りにもキビシイ憲法です。そのわりには、毎日の宴会は楽しかったけど……。だから、質問にピッタリの答を教えてしまいましょう。

まず、レジスタを左のように対応させると、答としては次ページのようになります。

8ビット長レジスタ		16ビット長レジスタ	
Z 80	8086	Z 80	8086
A	AL	BC	CX
B	CH	DE	DX
C	CL	HL	BX
D	DH	SP	SP
E	DL	PC	IP
H	BH	IX	SI
L	BL	IY	DI

Z 80と8086のレジスタの対応

```
        ⋮
        CALL    EXX
        ⋮

EXX     PROC
        XCHG    BX,CS:REGBX
        XCHG    CX,CS:REGCX
        XCHG    DX,CS:REGDX
        RET
EXX     ENDP

REGBX   DW      0
REGCX   DW      0
REGDX   DW      0
```

と、一般的にはこうなりますが、もし、Z80で使用していないレジスタが1つでもあれば、メモリではなくレジスタと置き換えることも可能です。ここでは例としてZ80のインデックス・レジスタIYを使っていない、すなわち、8086ではDIレジスタが空いていると仮定してマクロ定義をしてみました。

```
EXX     MACRO
        XCHG    BX,BP
        XCHG    CX,DI
        PUSH    ES
        MOV     ES,DX
        POP     DX
        ENDM
```

余談ですがZ80の「EX　AF,AF'」は、フラグを無視すれば「XCHG　AL,AH」で置き換えが可能です。ところで、星の砂ってどうしてあんなにきれいな星の形をしているんでしょうか。一説によると、ヨロンのあの満天に散りばめた星の群れから、時々はみ出して落ちてきたという話ですが、本当のことを知っていたら教えてください。星降る乙女さま……。

13 INC命令とキャリーフラグ1

聞いてください。わたしの悩み。マシン語の便利な用法がたくさんでている本を買ったんです。その本にはループ命令には、LOOP、LOOPZ、LOOPE、LOOPNZ、LOOPNEがあると書いてありました。早速、わたしは役立てようと思いました。
SIレジスタに、メモリ・ブロックの先頭アドレスを格納して、このメモリ・ブロックの中に0があるかどうかサーチせようとしたんです。

```
        ⋮
LP001:  CMP       BYTE PTR [SI],0
        INC       SI
        LOOPNZ    LP001
        JNZ       SHORI1
        ⋮
```

これが、その時のプログラムです。でも、メモリ・ブロックに0があっても、いつも最後までLOOPしてしまうんです。このプログラムにはバグでもあるんでしょうか。

悩める美人のOL（福岡）

答

オーッと、美人のOLとはウレシイじゃありませんか。マシン語というのは、蒸気機関車みたいな人間臭さがあるせいか、あまり女性には好かれませんでした。でも、女性がマシン語を操るというだけで、パソコンのイメージが‘ネクラ’から‘ネアカ’に変わりそうです。

さて、このプログラムは一見すると正しいようですが、実はそこに大きな落し穴が潜んでいます。それは、フラグというものに対する過信です。フラグレジスタというレジスタは、思ったより好き嫌いの激しい性格をしています。

ここで問題となるのは、「LOOPNZ」の命令が実行されるまでのフラグ変化です。おそらく美人OLさんは、「INC　SI」のフラグ変化を忘れていたのでしょう。8086では、16ビット長の「INC」、「DEC」命令であってもフラグが計算結果にしたがって変化するのです。このような間違いは、Z80などのマシン語か

ら、8086へと移行したプロフラマーの初期の段階によく現れる代表的なものですから注意してください。

ここでやった「INC　SI」という作業は、SIが0でない限り、常にゼロフラグをクリアすることになりますから、最後までループしてしまったというわけです。

では、どうしたらいいかというと、CMP命令の前で「INC　SI」を実行すればよいのです。

```
              ⋮
          DEC         SI
LP001:    INC         SI
          CMP         BYTE PTR [SI],0
          LOOPNZ      LP001
          JNZ         SHORI1
              ⋮
```

もし、AX、DI、ESレジスタが空いていれば、8086特有の命令を使って次のようにすることもできます。

```
          MOV         DI,SI
          MOV         AX,DS
          MOV         ES,AX
          CLD
          XOR         AX,AX
          REPNZ       SCASB
          JNZ         SHORI1
              ⋮
```

マシン語も女性に好かれるようになると、もっともっと夢のある世界が大きく広がるんですが……。まずは‘悩める美人のOL’さんの悩みを解消したことで、少しは未来が明るくなった気がしませんか。

14 ＋／－を交互に

オス。おいどんは、薩摩隼人でごわす。時代の波に乗り遅れぬよう、おいどんもパソコンを買ったでごわす。BASICは、ハッキリいって遅くてイライラする。やはり、時代はマシン語でごわすな。

おいどんは、未来を的確に見つめるため、いまあるプログラムを作っておるのだが、ある事情でコールするたびに＋50と－50という値を交互に返してくれるようなプロシージャが必要なのでごわす。現在は次のようにしておるが、どうもこのプログラムはスッキリしないのでごわす……。

```
MOV_AX      EQU     0B8H

DAT50       PROC
            DB      MOV_AX
DATKP       DW      50
            CMP     AX,50
            JE      DAT51
            MOV     AX,50
            JMP     DAT52
DAT51:      MOV     AX,-50
DAT52:      MOV     DATKP,AX
            RET
DAT50       ENDP
```

（注）DS＝CSと仮定する

ぜひ、的確な評価をしてほしいでごわす。お願いでごわす。

マシン西郷（鹿児島）

答

これはこれは、時代を超越したような質問でごわすな。こちらまで影響を受けてしまいそうでごわす。

このプログラムは、確かに見るからにスッキリしませんね。プログラムを書き換えながら使うなど、苦労の跡はうかがえるのですが、苦労がむくわれてな

いようです。でも、プログラムがスッキリしないと感じている点に、マシン語の夜明けを感じます。

マシン語プログラムで、スッキリしないとか不格好だと感じた時は、命令の一覧表（三の章の8086ニモニック表）を見直すと、いい知恵が浮かぶことがあります。それは、マシン語を覚えたてのころは、使い慣れたニモニック以外使わない（使えない）という傾向が強いからです。

この西郷クンも、便利な命令をまだ見落としているのです。こんな時には、「NEG」という最適の命令があるんですよ。

```
DAT50   PROC
        MOV     AX,DATKP
        NEG     AX
        MOV     DATKP,AX
        RET
DAT50   ENDP

DATKP   DW      50
```

（注）NEGはメモリに対しても有効

「NEG」とは、NEGATE（否定する、正負を反転する）の略ですが、この命令を使えば一発で正なら負へ、負なら正へと変換してくれるのです。もちろん、ここでの正とか負というのは、問2にあったように、あくまでもプログラムを組む人の感覚であることを忘れないでください。一応サインフラグ上では、ビット15（8ビット長であればビット7）の値で＋／－を判断するようになっていますが……。

時計を読むのに、何時何分過ぎと読むか、何分前と読むか、それは読む人の勝手というようなものです。では、同じような感覚で0と何かを交互に返すプロシージャ、と問題を変えたらどうでしょうか。例えば、0と50を交互に返してくれるようなプロシージャがほしい場合です。もちろん、質問にあるようなプログラムにもどってしまうようでは、維新の夜明けは遠くなるばかりです。

```
DAT50   PROC
        MOV    AX,DATKP
        XOR    AX,50
        MOV    DATKP,AX
        RET
DAT50   ENDP

DATKP   DW     0
```

（注）XORはメモリに対しても有効

今度は論理演算XORです。XORは二度繰り返すと値が元にもどるという特徴を利用するわけです。では、初期値を0ではなく別の値にしたらどう変化するか、XORに対する理解を深めるためにも自分で確認してみてください。

よくわからないという人は、キチンと二進数にして演算すればいいでしょう。ナニ、そんな簡単なこと知っていたでごわすか!! こりゃ、失礼したでごわす。

15 MOV命令を減らす

わては大阪の生まれだす。手紙も電話も大阪弁以外ではしませんのや。もちろん、英語だって大阪なまりだす。それが、浪花のド根性と思うとんのや。
そ、そのわてが、なんとマシン語に凝ってしもたのや……。くやしいけど、わてのパソコンかて大阪弁は理解してくれへん。これはホンマに残念なことやで。マ、それはいいとして、わてのマシン語というと、ムーブ（MOV）命令の連続なんや。
やっと、ワークエリアが自由に使えるようになったんやが、その中身を操作するたびにムーブ、ムーブや。例えば、こんな具合いや。

```
MOV    AX,OKANE
SUB    AX,2
MOV    OKANE,AX
MOV    AX,YAKSO
INC    AX
MOV    YAKSO,AX
```

「OKANE」とか「YAKSO」というのがワークエリアなんやけど、アドレスでいうたらお隣りどうしやで。もっと、わての大阪弁みたいに流ちょうにいかんのやろか。ア、プログラムの内容はお金を使って薬草を買うた、という意味や……。

浪花ぼてじゃこ物語（大阪）

答

英語いうたかて世界中どこでも通用するわけではあらへん。わてもイタリアを放浪しよった時は、大阪弁使いよったで……デタラメの……。

外国人というと、みな英語を話すと思ってしまうのが島国日本人の悲しさ。苦労して下手な英語を使っても、通じない国はたくさんあります。どうせ通じないのなら、日本語でも同じこと。真剣に話すと、結構相手もわかってくれるようです。

その日本語でも流ちょうに話すのは難しいのに、マシン語を流ちょうに使いたいとは驚きです。凡人には、なかなか言えないことです。

では、さっそく流ちょうでないプログラムを見てみましょう。流ちょうに見

えない原因は、アドレスをすべてダイレクトに指定しているからです。忘れないでください、インデックスレジスタというアドレスを示せるレジスタがあることを。

かりに、「OKANE」と「YAKSO」というアドレスが、番地の若い順に並んでいるとしましょう。一方のアドレスを指定すれば、隣りのアドレスは簡単にわかります。流ちょうな日本語ができなくたって、隣りの家をいちいち住所で言う人はいませんね。どこか一軒を指定すれば、あとは一軒先とか手前というように相対的に表現するのが普通です。

```
MOV   SI,OFFSET OKANE
SUB   WORD PTR [SI],2
INC   WORD PTR [SI+2]
```

これで、プログラムに要するメモリ数は約半分になりました。気になったムーブ命令も1つになりました。……が、ムーブ命令の多少とプログラムの良否については、まったく関係ありません。

そのことより、メモリ中のデータを操作する場合、できるだけアドレスをレジスタで指定するように心がけることが大切でしょう。ほとんどの場合、それだけでメモリ数／クロック数の減少につながります。

ところで、レジスタAXは他の汎用レジスタ(BX,CX,DX)よりも、メモリからのデータのロード、またはデータのストアに対するクロック数が少ないので、最後の「INC　WORD PTR [SI+2]」は、むしろ元のままの(MOV)命令を使ったプログラムの方が、実行スピードが早いくらいです。

また、関連するワークエリアのアドレスが前後に散らばっていたりする場合にも、インデックスレジスタ(SI/DI)を使用するとプログラムはスッキリします。もし、あなたがMASMを使っているなら、メモリのワークエリアを構造体として宣言しておくとプログラムは、さらに見やすくなります。

どないだ。これで、マシン語も大阪弁に一歩近づいたでっしゃろ!!

16 ビット転送

ニッポンのみなさ～ん。お元気ですか……。
ワタシはおじいさんとおばあさんが日本人の日系三世です。英語名はドナルド、日本名は拓です。だから、ドナルド・タクというわけです。もちろん、通称はドナルド・ダックです。でも、初めて日本へ行った時'ドナルド'って呼ばれても誰のことかサッパリわかりませんでした。それに、日本語でいう『マクドナルド』というお店、これも行ってみるまで何のお店かわかりませんでした。ヤッパリ、ワタシはアメリカ人です。
ところが、うれしいことに日本人もマシン語を使うというじゃありませんか。それならというわけで、ワタシもマシン語を始めました。だから、まだ入門者です。そこで、ちょっと教えてください。
AL レジスタのビット5だけを、あるメモリにそのまま移したいのです。ビット5以外は変更したくありません。いまは次のようにしています。

```
        MOV   BX,OFFSET MEMRY
        TEST  AL,00100000 B
        JE    RSET 5
        OR    BYTE PTR [BX],00100000B
        JMP   XSET 5
RSET5:  AND   BYTE PTR [BX],11011111B
XSET5:          ⋮
```

ヤッパリ、こんなもんでしょうか。

Donald Tack（ロサンゼルス）

答

実は、私も憧れのディズニーランド（東京ではなくロスのほうです）へ行く前日、現地の人に明日はドナルドダックに会いたいと言ったのですが、いくら「ドナルド」「ドナルード」「ドーナルド」と言ってもわかってもらえなかったことがあります。

あれを『ダーナーダック』って発音するなんて……。『マクダーナー』というお店も日本にはないですからね。カタカナの英語がこんなにデタラメだとは、その時まで知りませんでした。ちなみに、私が話した現地の人というのは、一

応日本語も話せる日系三世の人だったんですが……。

こうなると、マシン語を国際語にするしか方法はなさそうです。質問にも快く答えてしまいましょう。

まず、質問にあるプログラムは、少し手直すだけで無駄なジャンプ命令が省けてスッキリさせることができます。

```
         MOV    BX,OFFSET MEMRY
         AND    BYTE PTR [BX],11011111B
         TEST   AL,00100000B
         JE     RSET5
         OR     BYTE PTR [BX],00100000B
RSET5:          ⋮
```

これだけで十分と言えないこともありませんが、この程度のことでジャンプ命令を使うのはカッコ悪いと感じる人もいるでしょう。そこで、ALレジスタの値はこわれてもいいという条件で、次のようにプログラムを組むのも一考です。

```
         MOV    BX,OFFSET MEMRY
         AND    BYTE PTR [BX],11011111B
         AND    AL,00100000B
         OR     BYTE PTR [BX],AL
           ⋮
```

「なんだ、変わりばえがしないな!!」なんて思いませんでしたか。確かに、この例に関してはそう言われても仕方ありません。では、もしもビット5とビット3の2つの値を移したいとなったらどうでしょうか。

最初のやり方では、プログラムが単純に倍になってしまいます。それに対して、二番目の方法なら各AND命令のオペランドを変更するだけで済みます。これはメモリ数や速度に影響を与えるものではありません。

つまり、両方を知っていればプログラムに合わせて効率のいい方法を選べるということなのです。やはり、何事においても基本は大切ですね。

英語の基本はABC……、決してカタカナではなかったのです。

17 プロシージャへのジャンプ

ボクの友人にパソコンを小学生の時から持っている上、すでにマシン語までマスターしてしまった者がいます。ボクは、1年前やっと高校入学と同時に買ってもらいました。そして、なんとかBASICは彼と同レベルになりましたが、マシン語ではまだまだその友人にかないません。そこで彼に内緒で質問します。

```
        ⋮
CALL        xxxx
RET
```

マシン語プログラムには、こういうケースがよくでてきます。ボクも最初はこのままでしたが、ある時コールしてからリターンするなら、「JMP xxxx」とすればいいのではと思ったのです。どうせ、コール先の RET 命令で元のルーチンへ戻れるからです。
以来、このような場合には「JMP xxxx」としてきましたが、それで問題が起きたことは一度もありません。でも、ボクの友人にその話をしたら、いずれダメな時も出てくると言います。理由を聞いてもハッキリ教えてくれません。
いったい、どういう時にダメなんでしょうか。ソッと教えてください。ボクは、今でも大丈夫だと思っています。

ドラＱ（新潟）

答

友人に自慢されている様子がよくわかります。似たような経験は誰しも一度はあるものです。でも、たったの1年でこんな鋭い疑問がわくのも、その友人のおかげと思えばいいじゃないですか。

この問題は、それほど奥が深いのです。メモリに余裕がある場合には、入口と出口は1つという原則を守ってプログラムを組みますから、質問のような手法はとりませんが、このような手法でも困ることはありません。それどころか、いったん、メモリ不足になった場合には、1箇所で1バイトの節約になりますから、こういう箇所を必死に捜すことになります。また、スピードのアップ(あくまでも計算上です）にもなっています。

では、どんな時に困るかというと、それはスタックを操作しながらプログラムが走っている場合です。例えば、メインプログラムからプロシージャ「AAAA」がコールされ、プロシージャ「AAAA」の最後でプロシージャ「CCCC」がコールされているとします。

```
MAINP:          ⋮
        CALL   AAAA
                ⋮
```

```
AAAA:           ⋮
        CALL   CCCC
        RET
```

```
CCCC:   ⋮
        RET
```

確かに、ここでプロシージャ「AAAA」からプロシージャ「CCCC」へジャンプすれば、プロシージャ「CCCC」のRET命令でメインプログラムへともどります。しかし、プロシージャ「CCCC」のプログラムが、条件によってはメインプログラムへ直接リターンするように設計されていたらどうでしょうか。

つまり、プロシージャ「CCCC」が「MAINP」から数えて常に二段目にコールされるようになっていて、条件によってはスタックを操作（スタックポインタを＋2）して「AAAA」へもどらずにダイレクトに「MAINP」へリターンするような場合です。もし「JMP　CCCC」でこのようなケースに出会うと、スタックが狂ってしまい「MAINP」を通り越してどこかへ暴走してしまうことになります。

もちろん、プログラムを作った本人がこのような作り方をしなければ問題はないのですが、共同で大きなプログラムを開発したり、他人のプログラムを借用したり、あるいはシステムの内部ルーチンをコールする場合など、こういった危険性はアチコチに潜んでいるのです。たとえ入口と出口は1つという原則にそってプログラムが開発されようと、完全に守られるという保証はありません。

具体的な例をあげてみましょう。画面上にピョンピョン跳ねている小さなボールが1つあるとします。ボールは画面に散らばっている釘に触れると破裂してしまいます。わかりやすくするため、設定はたったこれだけです。

プロシージャ「AAAA」はボールとすべての釘との衝突を管理するルーチンで、ボールの動きを管理している「MAINP」からコールされています。「AAAA」では、BXレジスタにボールの座標、CXレジスタに釘の座標を入れ、次々と座

標チェックと破裂を実行するプロシージャ「CCCC」をコールします。

```
AAAA  PROC
      MOV   BX,BALXY
      MOV   CX,KUGZ1
      CALL  CCCC
      MOV   CX,KUGZ2
      CALL  CCCC
      MOV   CX,KUGZ3
      CALL  CCCC
        ⋮
      RET
AAAA  ENDP
```

```
CCCC  PROC
      CMP CX,BX
      JE    CNRT
      RET              ←AAAAへもどる
CNRT:
        ⋮              ←破裂のプログラム
      POP   AX         ←ダミー
      RET              ←MAINPへもどる
CCCC  ENDP
```

プロシージャ「CCCC」では、ボールと釘との衝突チェックを行いますが、ひとつでも釘とボールが衝突していたら、その時点でボールは破裂し残りの釘との衝突チェックは不要になります。つまり、プロシージャ「CCCC」からプロシージャ「AAAA」へもどるのではなく、直接「MAINP」へもどるケースが生じるわけです。これを実現しているのが、破裂処理のあとにあるダミーのスタック操作です。このスタック操作が成立する前提として、スタックの条件（レベル）を統一するようなプログラムが要求されているわけです。

面倒そうですが、いずれプログラムに慣れてくると、スタックを考えながらプログラムを組むようになるのです。新潟のドラQ君がそうなるのも時間の問題です。もう友人にも自慢されなくてすむでしょう。

18 命令の書き換え

我輩は猫である。名前はまだない。猫に小判という諺はあるが、猫にコンピュータという諺はない。だから、猫がコンピュータに興味を持ってもおかしくはない。

……という書き出しで始まる『猫とコンピュータ』という小説を知っていますか。おそらく、まだ誰も知らないはずです。というのは、まだ完成していないからです。私がその作者ですから、それは間違いのない事実です。

実は、この猫は超能力を持った猫で、プログラマーである飼い主に、プログラムのバグや欠点を教えるというのが話の大筋です。しかし、作者である私がプログラムの初心者なので、欠点を直すというのが大変なことなのです。それが、小説が進行しない最大の原因となっています。

いま、小説の中のプログラマー氏が次のような2つのプログラムを組みました。その2つはほとんど同じ内容のプロシージャで、まん中の一部が違っているだけです。どのような助言を猫が与えるべきでしょうか。

```
ADDA1   PROC
          ⋮
        INC   AX
          ⋮
        RET
ADDA1   ENDP
```

```
SUBA1   PROC
          ⋮
        DEC   AX
          ⋮
        RET
SUBA1   ENDP
```

いま考えているのは、前半と後半をプロシージャとして共通化する方法ですが、そうすると前半のプロシージャの途中でリターンしたら困りそうです。

ペンネーム猫目漱石（愛媛）

答

岡目八目ならぬ猫目八目というわけですか。しかし、猫の先生にあたる作者がプログラムをよくわからないとなると、超能力猫からバグ猫に……。下手をすると、化け猫小説となってしまいそうです。

そこで、質問にある修正案通りにプログラムを組んだことにして、それが超能力猫の判断に見合うかどうか、まずチェックしてみましょう。

```
ADDA1   PROC
        CALL   HALF1
        INC    AX
        CALL   HALF2
        RET
ADDA1   ENDP

SUBA1   PROC
        CALL   HALF1
        DEC    AX
        CALL   HALF2
        RET
SUBA1   ENDP

HALF1   PROC
          ⋮
        RET
HALF1   PROC

HALF2   PROC
          ⋮
        RET
HALF2   ENDP
```

「HALF1」というのが前半の共通プロシージャで、「HALF2」が後半の共通プロシージャというわけです。一見すると、うまくまとまったような気がしますが、問題は「HALF1」のプログラムの途中でリターンすることがある場合です。

つまり「HALF1」の途中でのリターンは、それぞれ「ADDA1」「SUBA1」からのリターンを意味していますから、この場合「HALF1」の中で特別にスタック操作をしてリターンしなければならないわけです。

リターンに関するスタック操作は問17にもありましたが、キチンとスタックの状態を把握さえしていれば難しいことではありません。しかし、プログラムの途中でかりに「RET　N」などとなっている場合、いったんどこかにジャンプしてからスタックを操作しますから、意外と面倒です。

さらに……、「HALF1」がプロシージャをコールしているとなると、話はますます複雑になります。というのは、そのプロシージャ先でスタック操作が行われている可能性だってあるからです。

……どうやら、このままでは化け猫小説になりそうな雲行きです。もっと簡単な手法で解決することにしましょう。それは、例の書き換えを使う方法です。

```
INC_AX   EQU    40H              ←40H=INC AX
DEC_AX   EQU    48H              ←48H=DEC AX

ADDA1    PROC
         MOV    CS: POINT,INC_AX
         CALL   ADSB1
         RET
ADDA1    ENDP

SUBA1    PROC
         MOV    CS: POINT,DEC_AX
         CALL   ADSB1
         RET
SUBA1    ENDP

ADSB1    PROC
           ⋮
POINT    DB     INC_AX
           ⋮
         RET
ADSB1    ENDP
```

こうすれば、プログラムを完全共有化できる上、スタックなどの余計な心配をする必要はなくなります。プログラムが長くなればなるほど、このような書き換えによるメモリ節約は効果を発揮してくれます。

SUBA1で「MOV　CS:POINT,DEC＿AX」実行後ADSB1をコールしています。一見すると、そのままADSB1へと流してもよいように思えますが、CALL命令によってCPU内部の命令キュウをクリアする働きも兼ねていますから、「無駄だ」などと、うっかり削ってしまわないように注意してください。

これで、めでたく『猫とコンピュータ』が完成するといいですね……。

問 19 フラグで合図を送る

以前、新潟のドラＱという者が質問をしたと思いますが、あれはボクの弟です。弟とボクは３歳離れており、ボクも去年は大学受験でした。入学と同時にパソコンを買ってもらったので、パソコンに関しては弟と同じレベルです。
ところで、先日弟がした問14への回答について、もう少し詳しく聞かせてください。あの時の回答には、スタック操作の例としてボールと釘との衝突チェックをあげていましたが、説明としては理解できても現実にはピンとこないのです。
それは、ボールと釘の衝突を判定するプロシージャ「CCCC」から、どうして「MAINP」へ直接もどるのかという疑問です。衝突チェックなんていうのは、プログラム実行時間からすれば瞬時のできごとです。残りの釘のチェックをしたところで、実際にはなんら問題は生じないはずです。
それとも、直接「MAINP」へもどることにより、なにか特別なメリットでもあるのでしょうか。兄として、知りたいと思います。

下宿先のドラＱII（山形）

答

ウ～ム。そこまで突っ込んだ質問がくるとは、こちらとしても不覚にも予測していませんでした。さすが、兄のドラＱIIさんです。

あの時の例は、あくまでもスタック操作の一例として出したのですが、今回の質問により問題は衝突チェックのテクニックへまで発展してしまったようです。確かに、実行時間についてだけ考えれば、残りの釘との衝突をチェックしたところで何の支障もありません。だとすれば、「MAINP」へ直接もどる必要もないということになりそうです。

そこで、まずは破裂があったかどうかを「MAINP」へ伝達する方法について考えてみましょう。最初に思いつくのは、破裂したことを示すワークエリアを確保し、そのワークエリアの値を「MAINP」でチェックする方法です。

おそらく「MAINP」の書式はこのようなものになるはずです。もちろん「BATWK」に破裂フラグを立てるのはプロシージャ「CCCC」の役目です。そして、このようなプログラムであれば「CCCC」から直接「MAINP」へもど

```
BATWK   DB      0

MAINP:          ⋮
        CALL    AAAA            ←衝突チェック
        MOV     AL,BATWK
        OR      AL,AL
        JNE     GOVER           ←破裂フラグが立っていればゲームオーバーへ
                ⋮
```

(注) DS＝CS と仮定する

る必要もありません。

つまり、プロシージャ「CCCC」は衝突があるないにかかわらず、すべての釘とのチェックをすればいいからです。では、どのような場合に直接「MAINP」へもどったほうがいいのでしょうか。それは「MAINP」が次のようになっている場合です。

```
MAINP:          ⋮
        CALL    AAAA
        JE      GOVER
                ⋮
```

プログラムとしては、見るからにこちらのほうがスッキリしています。破裂を示すフラグはゼロフラグですから、ワークエリア「BATWK」も不要です。もちろん、プロシージャ「CCCC」ではダミーのスタック操作後にゼロフラグを

```
CCCC    PROC
        CMP     BX,CX
        JE      CNRT
        RET                     ← BBBB へもどる
CNRT:
                ⋮               ←破裂のプログラム
        POP     AX              ←ダミー
        XOR     AX,AX           ←ゼロフラグを立てる
        RET                     ← MAINP へもどる
CCCC    ENDP
```

立てなければなりませんが。

つまり、残りの釘とのチェックを省くことで、衝突の判定に利用したゼロフラグをそのまま破裂フラグとして利用しているわけです。もしも、最初のような書式であれば、ゼロフラグを立てる代わりに、「BATWK」に1を入れるというプログラムが必要です。

今回はゼロフラグでしたが、このほかにもプロシージャからプロシージャへの条件の引渡しには、キャリーフラグなどもよく使用されます。フラグを単にその場の条件分岐のためだけでなく、引数代わりに活用することで、プログラムの雰囲気はかなり変化してきます。ドラQ兄弟にも、あるいは少しばかり差がついてしまったかもしれません。

20 範囲のある比較

ガーガー……。エー、聞こえるあるか。どうぞ……。

こちらは地球から遠く離れたアンドロメチャ星雲、メソポチャ星の外星広報担当、ルートカ・ダッペ将軍ある。ヨロシクあるネ。現在、銀河太陽系惑星・地球に向け知能情報収集のため超能力波を送信中。地球の名誉のため答えるあるよろし……。

これまでの情報によれば、地球にもようやくコンピュータが発生したようであるが、まだまだマシン語は普及してないとのことあるね。わがメソポチャ星は、マシン語が唯一の言語あるが、そのレベルは不明ある。

例えば、AL レジスタの値が 5～20の範囲にあるかどうかを調べる場合、われわれは次のように話すあるネ。フラグの使い方の見本みたいあるよ。AL レジスタがその範囲にない時は、キャリーフラグを立てて結果を返すようにしているある。

```
CHECK    PROC
         CMP    AL,5
         JNB    CHENR
         RET
CHENR:   CMP    AL,21
         JNB    SETCY
         CLC
         RET
SETCY:   STC
         RET
CHECK    ENDP
```

地球では、どんなもんあるか。教えるよろし……ガー……。

ルートカ将軍（メソポチャ星）

答

ワッ!!

いきなり変な質問が私の頭に飛び込んできたある……。どうやら、地球人としての能力を調査されているみたいあるネ。これは、真剣に考えなくては……。

しかし、プログラムのレベルからすると、入門者と上級者の中間あたりをウロついているような星のようです。というのは、結果をキャリーフラグで返す

という高級なテクニックを使っているわりには、プログラムがどうも低レベルなのです。

確かに、キャリーフラグのセット／リセットなど、基本的な常識も知ってはいます。ところが、それらのテクニックがプログラムに反映されていないのです。少なくとも、このプログラムは次のようにするべきです。

```
CHECK   PROC
        CMP   AL,5
        JB    CHERT
        CMP   AL,21
        CMC
CHERT:  RET
CHECK   ENDP
```

せっかくニモニックには「CMC」というキャリーフラグを反転する命令があるのですから、ここで使わないという手はありません。なかなか「CMC」を使うチャンスなんてないですからね。ここは絶好の使用例といえるでしょう。

さらに、ALレジスタが破壊されてもいい場合には、マイナスの数値はプラスの大きい数値と同じ(問2参照)という、マシン語数値独特の特徴が活用できます。

```
CHECK   PROC
        SUB   AL,5
        CMP   AL ,21-5
        RET
CHECK   ENDP
```

最初に5を引いてしまうことにより、5未満の数（0〜4）は−5〜−1、つまり十六進数ではFB_H〜FF_Hとなります。これは結局21以上（5を引く前）の数値ですから、一度のCMP命令で範囲のある判定ができるわけです。

ただし、この判定ではキャリーフラグの立ち方が最初の例と反転しています。このプログラムをコールしているほうでも、条件分岐の条件を反転させる必要があるのは言うまでもありません。

こうしてみると、地球のマシン語レベルも国際的、いや宇宙的に通用しそうなレベルにあるではありませんか。どうであるか、ルートカ・ダッペ将軍……？

21 レジスタペアの値を2倍に

拙者の名は、武蔵。名前が古いせいか、どうも話し方まで古くさいが、その点はご勘弁願いたい。
現代の武蔵はパソコンをこよなく愛し、マシン語を自在に操ろうと日夜キーボードを叩いておる。先んずれば人を制す。これも、永遠のライバル小次郎に先を越されないための知恵である。
とはいえ、現代の小次郎もどこかでパソコンをいじっているに違いない。いずれマシン語で対決する時が来るであろう。時代は違っても、武蔵が小次郎に負けることは許されぬことなのじゃ。
拙者はいま、対決用プログラムの中で BX,CX をペアとする32ビット長の値を2倍にしなければならんのだが、どうもうまくいかん。どうしたものであろうか……。海が荒れよるのォ。

武蔵旅情（玄海灘）

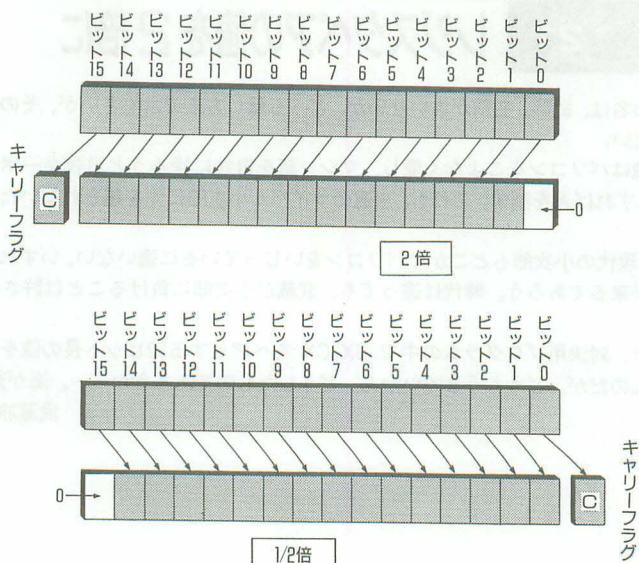
答

パソコンは人を選ばない。しかし、マシン語を話さない者には本当の心を開かないという。1台のパソコンをめぐって、マシン語とマシン語の壮絶な戦いが始まる……。

戦いに勝つためには、限られたニモニックの中で、最大限の工夫とヤリクリをしなければならないのです。勝負の世界のキビシサ、それは剣がマシン語に変わっても永遠に変わらぬ心理です。

ということで、BX,CX レジスタをペアとする32ビット長の値を2倍にするのは加算命令を使えば簡単に実現できます。

```
ADD   CX,CX
ADC   BX,BX
```

しかし、レジスタの値を2倍するというのは、なにも同じレジスタ同士を加えるだけではありませんね。そうです。左方向へシフトすればいいのです。その逆に右方向へシフトすれば、レジスタの値は半分になります。

ただし、残念なことに16ビットレジスタのシフト命令はあっても、32ビットレジスタを一度にシフトする命令はありません。そこで、この出っぱったキャリーフラグを利用するのです。

BX,CX レジスタをペアとする32ビット長の値を2倍にする

```
SHL    CX,1
RCL    BX,1
```

BX,CX レジスタをペアとする32ビット長の値を1/2倍にする

```
SHR    BX,1
RCR    CX,1
```

たったこれだけのことですが、それぞれのシフト命令の持っている役割に注意してください。このような答は簡単であればあるほど利用価値があるというわけです。

でも、これのどこが対決プログラムになるんだろうか……？

22 マイナスの値を1/2にする

遅いぞ、武蔵!!
もはや、拙者の対決プログラムは完成しておる。あとは武蔵の到着を待つばかりなのだが、拙者は到着が遅いからといってイライラしたりはせぬ。拙者だって、過去の話くらいは知っておるからな。
それにしても遅い……。まさか逃げたのでは。とりあえず、最後のプログラムチェックだけでもしておこうか。
ヤヤヤッ!! 暴走してしまった。まだ、プログラムにバグがあるようだ。どうもデータを半分にする部分がマズいようだ。本に載っていたのを、そのまま信用して使ってみたのだが、あの本にはバグでもあるのかも知れぬ……。
拙者の場合、マイナスのデータを半分にしただけなのだが、たしかマイナスとはプログラムを組む者が数値をどう見なすかという問題のはずだ。だから、そこにバグがあるとは思えぬし……。
ちなみに、そのマイナスのデータはAXレジスタとDXレジスタをペアとする値についてであるが、次のように本の通りにしている。ウ～ム……わからん。

AX,DXレジスタをペアとする32ビット長の値を1/2にする

```
SHR    AX,1
RCR    DX,1
```

もしかすると、あの本は武蔵の謀略本なのだろうか……。

ツバメ小次郎（巌流島）

答

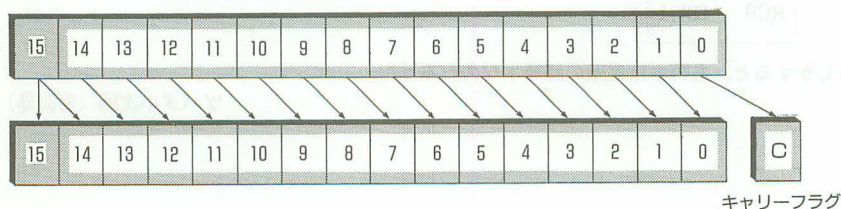
すでに武蔵と小次郎の心理戦は始まっているようです。ただ、どちらもプログラムが未完というのが気になりますが……。それにしても、あの巌流島でマシン語とマシン語が火花を散らすなんて、ワクワクしてしまいます。

さて、このバグは謀略によるものなどではありません。数値をマイナスと見なすか、プラスと見なすか、それは確かにプログラムを組む人の勝手です。しかし、数値を1/2にする場合はプラス／マイナスをキチンと区別しなければなりません。問21の方法は、あくまでも数値をプラスと見なしている場合に有効な

のです（2倍にするほうはプラス／マイナスを問わない）。というのは、マイナスと見なした数値の割り算（割る数＝2以上の場合）では、必ず最上位ビット＝1としなければならないからです。わかりにくいので、具体的な例で確認してみましょう。

AL＝11111100Bを　252と見なした場合：　AL÷2＝　126（01111110B）
AL＝11111100Bを　−4と見なした場合：　AL÷2＝　−2（11111110B）
AL＝00000100Bを　4と見なした場合：　AL÷2＝　2（00000010B）
AL＝00000100Bを　−252と見なした場合：　AL÷2＝−126（10000010B）

したがって、数値を常にマイナスと考えるのであれば、1/2したあとで最上位ビットを1にするか、最初に「STC」としてキャリーが最上位ビットに入るようにローテートすればいいわけです。しかし、数値のプラス／マイナスをサインフラグ（最上位ビット）に従って判断しようという場合は、最上位ビットの値が1/2後もそのままになるようにしなければなりません。

符号付きの1/2計算

こうすれば、−128～127（1バイトの場合）、−32768～32767（2バイトの場合）の数値をプラス／マイナスを問わずに1/2にシフト演算できます。そして、マシン語にはまさにこのための命令（SAR 命令）が最初から用意されているのです。つまり、プログラムを次のように訂正することで、問題のバグからは解放されるでしょう。

AX,DX レジスタをペアとする32ビット長の値を1/2にする

```
SAR    AX,1
RCR    DX,1
```

どうやら、武蔵も小次郎も問題が解決した模様です。私はミーハーですから、マシン語とマシン語の凄惨な対決を早くテレビで見たいものです。

23 かけ算を分解して高速化その１

CQ2メーター、CQ2メーター……。どなたか応答願います。こちらは難破船であります。現在地不明。乗務員１名。
初のパソコンによる自動操舵システム実験のため、栄光の出港をしてからどのぐらいの時がたったのでしょう。残された電源は太陽電池だけとなり、日没ともに自動操舵システムは波まかせとなってしまいます。
おまけに、本船のパソコンによる計算ルーチンにはバグがあったのです。ソースリストを見ると、かけ算ルーチンが空白のままなのです。おそらく、プログラマー氏があとから書くつもりだったのでしょう。コメントとして……、

```
BX=BX×144
```

と、なっていました。これは、BXレジスタの値を144倍するというような意味のはずです。でも、それ以上はわかりません。どなたか、このかけ算ルーチンをマシン語で作ってください。このままでは、いずれ幽霊船になってしまいます。
こちらのコールサインは「JJ1VRQ」です……。

難破船長パフェ（太平洋）

答

「JJ1VRQ」……？

ち、ちょっと、そりゃ私のコールサインですゾ。まだ一度も使ったことがないのに、どうしてそんなところに……。難破船にナンパされてしまった!?

これは、真剣になってコールサインを取りもどさなければいけません。ニモニックの一覧表（インストラクション表）には、かけ算の命令があります。しかしながら（MUL）や（IMUL）では使用されるレジスタが固定されていますから、そこのところを注意して使わないと、バグを生じることになります。

```
MOV   AX,144
MUL   BX
MOV   BX,AX
```

これが、もっとも単純に考えられるかけ算です。ここで、この計算結果がAXレジスタとDXレジスタに帰されることに注意してください。

ところで、これでも正しい結果は得られますが、(MUL)や(IMUL)を使用したかけ算では、実行クロック数がかなり長くなってしまいます。

早い話、これは実行スピードが遅いということです。そこで、144という数字を次のように分解します。

144＝2×2×2×2＋2×2×2×2×2×2×2

レジスタの値を2倍にするのは簡単です。それを繰り返せば、2倍が4倍、4倍が8倍と、ガマの油売りの前口上みたいに倍々にふくらんでいきます。これをプログラムで実行するのです。

```
SHL    BX,1      ← 2倍
SHL    BX,1      ← 4倍
SHL    BX,1      ← 8倍
SHL    BX,1      ← 16倍
MOV    AX,BX     ← AX＝16倍したBXの値
SHL    BX,1      ← 32倍
SHL    BX,1      ← 64倍
SHL    BX,1      ← 128倍
ADD    BX,AX     ← 144倍（128倍＋16倍）
```

かける数の決まったかけ算は、このように2のべき乗の和に分解してから計算すると、実行時間が大幅に短縮します。この効果は、かけ算をする回数が増えれば増えるほど大きく現れてきます。

では、早くプログラムを直して、「JJ1VRQ」を返してください。

24 かけ算を分解して高速化その2

超能力者……。信じられないかもしれませんが、ボクは超能力者です。テレビで有名なスプーン折りができるのです。
ただ、まだ超能力が弱いのでスプーンを1本折るのに3時間はかかります。それに、その間はスプーンを撫でながらズーッと念を入れなければならないので、とにかく非常に頭が疲れます。
そこで考えたのが、この念をパソコンにさせるということです。つまり、ボクがスプーンを折ろうとしている時に考えていることを、パソコンにそのままやらせればスプーンが折れるはずだと思うのです。
とりあえず、その準備プログラムとして必要なのが、AL×0A0$_H$の値をDIレジスタに入れるというかけ算プログラムです。現在は次のようにしています。

```
AXBHL:    MOV    AH,0A0H
          MUL    AH
          MOV    DI,AX
          RET
```

かけ算としては平凡なプログラムと思いますが、もう少し速度を速める方法はないものでしょうか。なんとかして超能力パソコンを実現したいのです……。

エスパー魔脳（宮崎）

答

超能力者も大変そうです。そんなに苦労しなくとも、両手を使えばスプーン折りくらい簡単に実現できるのに……。

どうせ超能力を使うなら、スプーンやフォークを割箸のようにタテに割るとか、普通の人にできないことをやってもらいたいものです。どうして超能力者はスプーンの首ばかりを折ろうとするのでしょうか。

……という疑問は別にして、このパソコンによる超能力はすごく楽しみです。本当に成功してもらいたいものです。

さて、この質問の例でも、実行速度を上げることは可能です。2のべき乗の和の例は問23でやりましたから、ここでは、100$_H$の倍数の和に分解する例を示しておきましょう。

$AL \times 100_H$

```
AX100   PROC
        MOV   AH , AL
        XOR   AL , AL
        MOV   DI , AX
        RET
AX100   ENDP
```

$AL \times 80_H = (AL \times 100_H) \div 2$

```
AX080   PROC
        MOV   AH , AL
        XOR   AL , AL
        SHR   AX,1
        MOV   DI , AX
        RET
AX080   ENDP
```

$AL \times 280_H = (AL \times 100_H) \times 2 + (AL \times 100_H) \div 2$

```
AX280   PROC
        MOV   AH , AL
        XOR   AL , AL
        MOV   DI , AX
        SHL   DI,1
        SHR   AX,1
        ADD   DI , AX
        RET
AX280   ENDP
```

$AL \times 0A0_H = (AL \times 100_H) \div 2 + ((AL \times 100_H) \div 2) \div 2 \div 2$

```
AX0A0   PROC
        MOV   AH , AL     2
        XOR   AL , AL     3
        SHR   AX,1        2
        MOV   DI , AX     2
        SHR   AX,1        2
        SHR   AX,1        2
        ADD   DI , AX     3
        RET              ─────────
AX0A0   ENDP              16クロック
```

この例ではトータルで16クロックですから、実に、60クロックもの節約となります。しかし、よほど速度を追求しているのでなければ、ロジックを考える

のに時間を取られるよりも、すなおに用意されている命令を使ったほうが得策かもしれません。

もし超能力パソコンが実現したら、ぜひそれでスプーンを縦に割ってください。首折りスプーンは、もう飽きましたので……。

25 CLDとSTD

わたしはコンピュータ占い師マーサ・カーイといいます。実は、コンピュータ占いといいましても、これまでのものはほとんどがイカサマでした。あえてネタをばらしてしまいますと、最初にインプットした占いデータを、ただ順番に出していただけなのです。
わたしは、職業がら女装をしていますが、本当は男性です。これも一種のイカサマなのですが、このほうがなんとなく神秘的に見えるのです。
ところが、先日イカサマを見破られてしまったのです。アラ、女装のほうではなくプログラムのほうですワ。ホホホ……。そこで、これからは本格的にコンピュータ占いをプログラム化しようと思い、覚えかけのマシン語でプログラムを組み始めました。
でも、データをブロック転送する際に、時々データがメチャクチャになってしまうことがあるのです。そのブロック転送は、5000_H〜$5FFF_H$番地の内容を5100_H番地へ転送するという簡単なものです。

```
CLD
MOV   SI,5000H
MOV   DI,5100H
MOV   CX,800H
MOV   AX,DS
MOV   ES,AX
REP   MOVSW
```

転送方向＋、SIレジスタに5000_H番地をセット、DIレジスタに転送先をセット、セグメントは同じデータ・セグメント内、そしてCXレジスタには転送ワード数をセットしています。どこも悪いところはないはずです。こればかりは占うこともできず、困ってしまいますわ、ホント。

ニューハーフ占い師マーサ（東京）

答

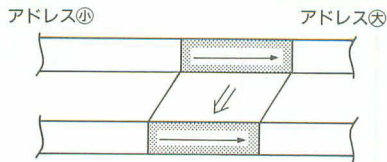
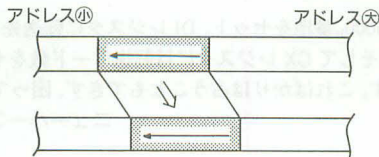
こういう丁寧な質問をいただきますと、こちらまで女装した気分になってしまいそうですわ、オホホホ……。

なんだか、おかしな空気が漂いだしたようです。このままでは、まともな回答ができなくなりますので、失礼ですが相手を見ないようにして答えることにします。

このブロック転送は、一見するとなんのバグもないように見えます。特に、バグがないと思ってしまうと、長いこと悩むことになるほどです。

しかし、ブロック転送において、転送するデータのあるアドレスとその転送先が重なっている場合は、データのディレクション・フラグ（転送方向を示すフラグ）をキチンと使い分けなければなりません。今回の場合も、このままでは5000_H番地の内容が5100_H番地に転送された時点で、まだ転送し終えていない5100_H番地の内容が消えてしまっているのです。

では、どのようにディレクション・フラグを使い分けるのか、図によって確認してみましょう。

STD命令（アドレスの大きい方から順次転送していく）

CLD命令（アドレスの小さい方から順次転送していく）

転送先がオリジナルのデータと重ならない場合、一般には＋方向とし、CLD命令でディレクション・フラグを０クリアします。それは、いちいち転送データのエンドアドレスを計算しなくて済むからです。そのせいでしょうか、ついつい－方向というのは忘れられがちな存在です。果ては、プログラムの先頭でCLDを実行しておいて、ストリング命令を使うたびに、いちいちディレクション・フラグを０クリアせずにすむようにプログラムを組む場合もあるくらいです。しかし、天災とバグは忘れたころにやってくる。それを忘れないでください。

```
STD
MOV    SI,5FFEH
MOV    DI,60FEH
MOV    CX,800H
MOV    AX,DS
MOV    ES,AX
REP    MOVSW
```

これで、ブロック転送は完璧に行われるはずですわ、オホホホ……。しまった、また相手の顔を見てしまったわ。ワワワ……。

26 CMP 命令でのジャンプ

わたしはナース。つまり看護婦さんです。でも、本当をいうとまだ看護婦見習いなんです。看護婦さんっていうと、白衣の天使なんて憧れる人もいますけど、実際はすごく大変な仕事です。

それはさておき、わたしは看護学勉強のためパソコンを活用しようと思っています。でも、そのためのソフトなど市販されていませんから、プログラムも全部自分で組まなければならないのです。プログラムはマシン語を使っていますが、不慣れのためバグがないということしか自信はありません。

ここに、AL レジスタの値（1〜5）によって 5 箇所に分かれるようなプログラムがあります。バグはないと思いますが、誰でもこう組むものでしょうか。

```
        CMP   AL,1
        JE    LABEL1
        CMP   AL,2
        JE    LABEL2
        CMP   AL,3
        JE    LABEL3
        CMP   AL,4
        JE    LABEL4
LABEL5:     ⋮
```

せめてプログラムだけでも見習いの肩書きが取れるとうれしいのですが。未熟なわたしのプログラムを、どうぞよろしくお願いします。

ホワイトエンジェル（佐賀）

答

ナース……。なんと美しい言葉のひびき。これでは、白衣の天使に憧れて、無理にケガをする人が続出しそうです。ついでに、マシン語のバグを治療してくれるような看護婦さんがいると、マシン語がもっともっと広がるのですが……。

それにしても、バグが出ないだけでは満足せず、さらに上を目指すというファイトはこちらが見習いたいほどです。

実は、このようなプログラムは初心者がよく作ります。しかも、プログラムとしてはバグはありませんから、このままズルズルと中級者になってしまうことも少なくないようです。しかし、条件分岐というのはプログラムの重要な拠点ですから、できるだけシンプルに、そして素早く分岐させることが大切です。

この例は、そういう意味では初歩の条件分岐といえるでしょう。ここでは分岐の条件にゼロフラグだけしか使用していませんが、せっかくCMP命令を実行したのですから、他のフラグも活用したいものです。

```
        CMP     AL,2
        JL      LABEL1
        JE      LABEL2
        CMP     AL,4
        JL      LABEL3
        JE      LABEL4
LABEL5:         ⋮
```

CMP命令を使う回数が4回から2回になりました。とりあえず、この程度の条件分岐ではこんなものでしょうが、分岐テクニックにはまだまだ種類があります。プログラムのレベルが上がるにつれ、分岐のレベルも合わせて上げていかないと、プログラムが条件分岐の山になってしまいます。

そんな時にこそマシン語専用のやさしい看護婦さんがいてほしい、と多くの孤独なプログラマーは願っているのです。

27 テーブルを利用したジャンプ

小話その１：ボクの友人は、50メートル競泳を見て「あれは身長競争だ」と言い張っている。ワケを聞くと「だって、身長50メートルの人間なら飛び込むだけでゴールだ」……。

くだらない……なんて言わないでください。こんな小話を作るのが、わが校自慢の小話クラブです。すでに、名作と自称するものがかなりあります。でも、大変なのはそれを管理することです。とはいえ、そこは高校のクラブです。なんと、名作小話をパソコンで管理することになったのです。
……ということで、プログラマーはパソコン所有者というだけの理由でボクの役目となりました。もちろん、カッコつけてマシン語でプログラムを組んでいます。
いま、0～99までの小話に簡単なアニメをつけようとしています。そのため、小話の番号別にそれぞれのルーチンへジャンプさせなければなりません。

```
CMP    AX,1
JB     ANI00
JE     ANI01
CMP    AX,3
JB     ANI02
JE     ANI03
   ⋮
```

こんな感じで各アニメ処理ルーチンへジャンプさせようとしていますが、どんなもんでしょうか。小話同様、名作と言われるようなプログラムにしたいと思います。

クラブ小話（和歌山）

答

同じような小話をひとつ……。

食事中に、母が肥満ぎみの父に向かって「理想の体重は身長から110を引いた値よ」と言った。それを聞いていた小学１年生の息子が、突然食べるのをやめた。ちなみに、息子の身長は110 cmだった……。

クラブ小話……なんだか、銀座のクラブと間違えそうな名前ですが、一応は

高校のクラブと信じておきましょう。でも、この質問には銀座のクラブ「小話」のほうが解決しやすいのです。というのは、テーブルを使うからです？？？

もちろん、ここでいうテーブルとはジャンプ用テーブルのことです。とりあえず、サンプルとしてテーブルには10のジャンプ先を用意してみました。

```
TABLE   DW    ANI00,ANI01,ANI02,ANI03,ANI04
        DW    ANI05,ANI06,ANI07,ANI08,ANI09

ATJMP:  MOV   BX,OFFSET TABLE
        ADD   BX,AX
        ADD   BX,AX
        JMP   CS:[BX]
```

これが、テーブルジャンプの一般的な用法です。ジャンプ先を増やすには、このテーブルにジャンプ先を追加するだけです。プログラムも短くて済む上、どのルーチンへジャンプするのも実行時間が変わらないというのが特徴です。

さて、プログラムの最後に「JMP　CS:[BX]」という命令があります。ラベル参照の場合JMP命令は相対ジャンプですが、この場合にはCS:[BX]に格納されているアドレスへとジャンプしてくれます。また、DS=CSであれば、セグメント・オーバーライド・プリフィックス（CS:）は省いてください。

28 一度のCMP命令で5つの条件判断

アー、忙しい忙しい。とにかく、この日は忙しい……。
わしか、わしはサンタクロースじゃよ。毎年、12月24日の夜には世界中の子供たちにプレゼントを配らなければならないのじゃ。そして、いつものことじゃが悩むのは配る順序についてじゃ。下手をすると夜が明けてしまうからな。
そこで、最近はコンピュータを導入して合理化を図っておるのじゃ。「エーッ!!」と驚く人もいるかもしれないが、このプログラム次第でわしの真価が問われるのじゃから、気合いも入ろうというもの。
いま、ここに5種類のプレゼントが用意されているとしよう。子供たちは、サンタ・プレゼント方程式により、皆ある数値を持っておる。そして、その値によりどのプレゼントがもらえるかが決まるようになっておるのだ。
その値とは、0、1、2〜7 F_H、80_H、81_H〜FF_Hの5種類で、ここからそれぞれのプレゼントルーチンへとジャンプさせるわけじゃ。テーブルを作ってジャンプさせるには不便だし、やっぱりCMP命令で1つひとつ分岐させるしか手はないかのう……。
とにかく、この条件分岐は何度も使用するので、できるだけ速く分岐させたいというのが、わしの希望じゃ。

サンタ日本代理人・黒須三太（富士山）

答

サンタの世界にもコンピュータが入り込んでいるとは、正直のところ驚かずにはいられません。そして、ここでも要求されているのはスピードです。まさに、スピードを制す者は世界を制すという感じです。

ちなみに、質問の内容をプログラムにすると次のようになっているのでしょうか。

```
          CMP    AL,1
          JB     SANT0
          JZ     SANT1
          CMP    AL,80H
          JB     SANT2
          JZ     SANT3
SANT4:           ⋮
```

これでも悪いということはありません。どちらかというと、非常に一般的といえるでしょう。しかし、ここはひとつ0、1、2～7F_H、80_H、81_H～FF_Hという、条件分岐のもとになっている値に注目してみたいのです。

この値が偶然に付けられたのか、それとも特別な意図があって付けられたのかは不明ですが、一度のCMP命令で5つの判定ができる実に有効な数値なのです。では、それぞれの値について、「CMP　AL,1」によるフラグの変化を示してみます。

	サイン	ゼロ	オーバーフロー	キャリー
0	1(NG)	0(NZ)	0(NV)	1(CY)
1	0(PL)	1(ZR)	0(NV)	0(NC)
2～7F_H	0(PL)	0(NZ)	0(NV)	0(NC)
80_H	0(PL)	0(NZ)	1(OV)	0(NC)
81_H～FF_H	1(NG)	0(NZ)	0(NV)	0(NC)

どうですか。4つのフラグをうまく利用すれば、一回のCMP命令ですべての分岐先へジャンプできそうですね。ただし、分岐の順序を間違えると、せっかく変化してくれたフラグが役に立ちません。次の例を参考に、注意してジャンプさせてください。

```
        CMP    AL,1
        JB     SANT0
        JZ     SANT1
        JG     SANT2
        JO     SANT3
SANT4:         ⋮
```

たった2バイト、4クロックの節約にしかなりませんが、要はその心意気が

大切なのです。10万回実行すれば、40万クロックの節約になるのですから。さらに、この分岐に優先順位（分岐する度合が多い方を先に分岐させる）をつけるところまで気を配れるようになると、スピード重視も本物です。もちろん、その場合でも先のフラグ変化表を確認するのは当然ですが……。

　せっかくあるフラグですから、数値を特に連続させなくてもいい場合は、このサンタ的分岐法を大いに活用しましょう。

29 共通項のあるジャンプ

私は、俗に団塊の世代といわれる中年のオッサンです。特にこれといった趣味はありませんが、まだまだ若いモンには負けるかという気持ちで、パソコンなども頑張ってやっております。

趣味はないと書きましたが、実は趣味に近いような形で学生のころから続けていることがあります。それは献血です。献血回数は、今のところ48回ですが、目標は100回を超えることです。

献血をすると、しばらくして血液の分析結果が送られてきますが、私はこのデータをパソコンにインプットして健康管理に役立てています。いま、ALレジスタの値（0、1、2～$7F_H$、80_H、81_H～FF_H）によって条件分岐するプログラムがあるのですが、ジャンプ先ではすべてALレジスタの値を8にしなければなりません。

現在のプログラムを簡単に書いてみますが、なぜか無駄があるような気がします。

```
        CMP     AL,1
        JB      PROG0
        JZ      PROG1
        JG      PROG2
        JO      PROG3
PROG4:  MOV     AL,8
           ⋮
```

以下、PROG 0～PROG 4すべて「MOV　AL,8」で始まります。無駄があるかどうか、そして改良できるかどうか見てください。

中年ベビー（兵庫）

答

同じようなタイプの人間は、アチコチにいるものです。私も、今では献血は趣味みたいなもので、血を抜いてもらうと気分がスッキリするほどです。それに、あのチクリと射された瞬間の痛みは、慣れると快感ですからね……？？？

ところで最近のニュースによると、献血された血液が無駄になることもあるそうですが、このプログラムにも想像通り無駄があります。それが、すべてのジャンプ先にある「MOV　AL,8」であるということは、誰でも簡単に想像が

つくでしょう。しかし、そんな簡単なことでも、いざ解決しようとすると、すぐには名案が浮かばないかもしれません。あるいは、そんなことは気にしないという人もいるでしょう。

でも、この本を読んだからには、プログラムの無駄は省くようにしてもらわなければなりません。

```
        CMP   AL,1
        MOV   AL,8
        JB    PROG0
        JZ    PROG1
        JG    PROG2
        JO    PROG3
PROG4:        :
              :
```

これで、ジャンプ先では「MOV AL,8」をする必要がなくなったわけです。この例のように、CMP命令と、それに対応するジャンプ命令との間にMOV命令のようなフラグに対して全く影響を与えない命令を挿入することがあります。また、ジャンプ命令がキャリーだけをチェックしてジャンプするのであれば、INC/DECのようにキャリーフラグに影響を与えない命令を挿入することも可能です。

プログラムを組んでいると、このような局面にはよく出会いますから、覚えておくと便利です。もっとも、CMP命令とジャンプ命令の間が、いたずらに長くなるのも、見やすさの点で問題があるとはいえますが。

30 スタックを利用したジャンプ

ホップステップジャンプで、こちらは南海の孤島で孤独な生活を送っているナゾの漂流者です。電気もない、ガスもない、水道もない、ナイナイずくしの中で、唯一の楽しみは頭を使った「想像マシン語」です。
時間だけはタップリとあるので、そのうち「想像マシン語」による「想像ゲーム」ができるはずです。でも、「想像デバッグ」を想像するだけでメゲそうです。
最近、覚えたばかりのテーブルジャンプを重宝して使っていますが、BX も SI も DI も使用中のルーチンがあります。だから、「JMP [BX]」も「JMP [SI]」も「JMP [DI]」もできません。ES レジスタは空いてるのですが、なんとか方法はないものでしょうか。

半魚人（沖の鳥島）

答

ち、ちょっと沖の鳥島といったら、少し前に話題になった水没寸前の岩じゃないですか。どうやって、そんな島へ……⁉

「想像マシン語」による「想像ゲーム」というのは初耳です。それが完成した時には、本物の人間コンピュータの誕生として大ニュースになるでしょう。

テーブルを用いてジャンプするのに、「JMP [BX]」も「JMP [SI]」も「JMP [DI]」も使えないということですが、メモリの参照として BP レジスタの存在を忘れているようです。ただし、BP レジスタのメモリ参照では、セグメントとして、暗黙に SS レジスタの値が使われますから、セグメント・オーバーライド・プリフィックス命令の（DS:）を忘れないようにしてください。また、メモリ参照のジャンプに限らず、レジスタにジャンプ先を格納してジャンプする方法もお忘れなく。テーブルジャンプの例としては、これらのジャンプのほかに PUSH 命令と RET 命令を組み合わせて行うこともできます。これは詰将棋でいうなら、ちょっとヒネった一手詰めといったところです。問27を例にしてプログラムを組んでみましょう。

```
ATJMP:  PUSH  BX
        MOV   BX,OFFSET TABLE
        ADD   BX,AX
        ADD   BX,AX
        MOV   ES,CS:[BX]
        POP   BX
        PUSH  ES
        RET                      ]←「JMP CS:[ES]」を実現するため
```

```
ATJMP:  MOV   ES,BX
        MOV   BX,OFFSET TABLE
        ADD   BX,AX
        ADD   BX,AX
        PUSH  CS:[BX]            ← JMP CS:[BX]を実現するため
        MOV   BX,ES
        RET
```

これはスタック操作の一種なのですが、どちらもスタックへジャンプ先アドレスを格納して、RET命令によりそのアドレスにもどるようにジャンプしています。条件としては、ESレジスタを破壊してもよいということですから、いずれにしても、ESレジスタの使い方が鍵になってきます。

第一の例では、ジャンプ先をスタックに入れるために使っています。そして、二番目の例では、ESをBXレジスタの保存用に使うことによって、BXレジスタを自由に使ってしまおうというわけです。

これらのテクニックを使うと、けっこう色々な形のジャンプができそうですね。ではそろそろ、救援船にジャンプしたらどうですか。天才半魚人さん!!

31 ビット別のジャンプ

映画ってスバラシイですね。映画を見るたびにそう思わずにはいられません。涙と笑いを誘いながら、夢と感動を与えてくれるのです。そして、映画館から外へ出た時の太陽のまぶしさと現実の鼓動、このギャップはテレビでは絶対に味わえないものです。
ところで、映画にはよく実在する町がでてきます。そして、映画の町こそが本物の町のような錯覚を覚えます。でも、日本にはカサだけを売っている傘屋さんなんてあるでしょうか。私は見たことがありません。
それなのに、『シェルブールの雨傘』の舞台は傘屋さんです。もし実在するなら、あんな町のあんなお店で傘を買ってみたい……。
そんなことを考えながら、プログラムを組むのが私の楽しみです。いま、ALレジスタにはビット別に意味を持たせた値が入っています。つまり、8種類のフラグになっているわけですが、これをビット別に各ルーチンへジャンプさせたいのです。

```
MAINP:  ROR   AL,1
        JB    PROG0     ←ビット0=1ならPROG0 へ
        ROR   AL,1
        JB    PROG1     ←ビット1=1ならPROG1 へ
        ROR   AL,1
        JB    PROG2     ←ビット2=1ならPROG2 へ
        ROR   AL,1
        JB    PROG3     ←ビット3=1ならPROG3 へ
        ROR   AL,1
        JB    PROG4     ←ビット4=1ならPROG4 へ
        ROR   AL,1
        JB    PROG5     ←ビット5=1ならPROG5 へ
        ROR   AL,1
        JB    PROG6     ←ビット6=1ならPROG6 へ
        ROR   AL,1
        JB    PROG7     ←ビット7=1ならPROG7 へ
        RET
```

ジャンプの優先権はビット0から順番に低くなっていきます。そして、このようにビット別にジャンプをさせるケースが数箇所あります。こういう場合はテーブルを利用したジャンプは無理でしょうか。

喫茶シェルブール（長崎）

答

シェルブールとはどんな町なのだろう。町には音楽があふれ、人々は色とりどりの傘をさしながら歩いている……。

『シェルブールの雨傘』を見た人ならそう思って当然ですよね。私も、その美しさに憧れて、わざわざシェルブールまで行ってきました。できたら、おみやげに傘でも買おうと思いながら……。

小さな港町には、にぎわいも音楽も傘屋もありませんでした。観光客もいない、ただの田舎町です。ひょっとすると会えるかも、と思ったカトリーヌ・ドヌーブ……いるわけがありません。

でも、なぜかホッとしました。みやげもの屋なんてあったらガックリきたでしょう。ただの町だからこそ、シェルブールは永遠にすばらしいのです。少し古い情報ですが、今でもそんな素朴な町だと信じています。

さて、このようなプログラムでもテーブルによるジャンプは可能です。

```
TABLE     DW     PROG0,PROG1,PROG2,PROG3
          DW     PROG4,PROG5,PROG6,PROG7

BTJMP:    MOV    BX,OFFSET TABLE
JUMP8:    MOV    CX,8
JMP8L:    ROR    AL,1
          JNB    NTJMP
          JMP    CS:[BX]
NTJMP:    ADD    BX,2
          LOOP   JMP8L
          RET
```

一見すると、これは質問にあったプログラムより複雑で、時間もメモリも余計に消費しそうな感じがするでしょう。それは事実であり、あえてこのようにする必要がない場合も多いかもしれません。

しかし、このプログラムの特徴は、複数のテーブルがあってもプログラムを共通して使用できる（JUMP8以下）という点にあります。また、ジャンプ先をテーブル上で管理できるので、変更やデバッグが非常に楽になります。

七色の雨傘ならぬ、八色のテーブルジャンプとして活用しましょう。

32 フラグ以外の条件ジャンプ

私は、いまだかつて「ウソとホラと坊主の頭はゆったことがない」マ界党公認の大政治家……岩内保羅夫（いわないほらお）でございます。政治家は「ウソつき、ホラふき、芸のうち」が当然でありますが、私はホラを言わない岩内保羅夫でございます。次回のマ界選挙で当選した暁には、マシン語を世界の公用語にしマ界の発展に寄与する所存でございます。
実は、ある宴会の席上、酔った勢いでついつい「フラグを見ないでゼロチェックをしてみせる」などと言ってしまったのですが、ハッと気付いた時には……ア～レマ「遅かりし由良之助」です。ホラを言わないどころか、私の口からはホラ、ホラ、ホラ……ホラの大連発です。
せめて、マシン語に関することだけでもホラじゃないようにしたいのですが、やっぱり無理ですかねェ。もし、ここで助けてくれたなら、当選の暁にはこの本を百万部ほど買い取ってあげます。それから、私がこれから売りだそうとしているホラ止めの薬「ホライワン」を百年分さしあげます。どうぞ、よろしくお願いします。

岩内保羅夫（マ界党）

答

どうもホラどころか坊主の頭までゆったことがありそうな雰囲気ですが、とりあえずマシン語についてだけはホラ吹きの汚名が晴れそうです。

ジャンプ命令をよく調べてみると、その中に「JCXZ」という命令があります。このジャンプ命令はゼロフラグをチェックしてジャンプするのではなく、CXレジスタが 0であればジャンプするという命令です。したがって、事前にゼロかどうかをフラグに反映させる命令を実行する必要がないのです。

例えば、SI レジスタで示されるメモリ内容がゼロの時にジャンプさせたい場合、メモリの値を CX へ代入するだけで、この命令が使えるのです。

CMP 命令によるメモリの 0 チェック

```
        ⋮
CMP   WORD PTR [SI],0     15クロック（3バイト）
JZ    ＊＊＊＊
        ⋮
```

JCXZ命令によるメモリの0チェック

```
        ⋮
MOV    CX,[SI]         13クロック（2バイト）
JCXZ   ****
        ⋮
```

メモリに対してINC/DECを使うこともできますが、これは実行速度、使用メモリ数ともに効率悪くなってしまいます。また、AND、OR、XOR、TEST、ADD、SUB命令による0チェックも可能ですが、やはりCMP命令を使ったほうが有利です。

その点、このJCXZ命令はCXレジスタへチェックしたい値を代入するだけですから、最も効率よく目的を果たすことができます。割り算をする時やセグメントレジスタの0チェックにも使えますから、うまく応用するといいでしょう。

ということで、マシン語のホラは解消されましたが、例の百万部の件やホラ止めの薬についてはアテにしないで期待することにします。

33 LOOP命令の特徴

みゃー……。
おみゃー、ナーゴヤいうたらエーところだでヨー。きしめんはウミャーし、ういろうもウミャーし、大阪と東京のエーところを取り入れた日本の中心都市だでヨ。それに、もうひとつの名物はパチンコ屋だで。
実は、わしゃパチンコが好きで好きで、そーれでナーゴヤに越して来たばかりの老人だでヨ。だから、これはテレビで覚えたインチキのナーゴヤ弁じゃ。テレビじゃ、ミャーミャー言うとったけど、実際にはそうでもないみたいだで。
それはそうと、わしのパチンコ仲間で出る台の番号をパソコンで研究しておるやつがおるでヨ。わーしはそのソースリストを持っとるでな、プログラムの無駄をなくそう思って一部直してやったでヨ。

```
            ⋮
LOOP1:    MOV   AX,[SI]
          CMP   AX.5
          JB    SUBRT
          CMP   AX,30
          CMC
          JNB   SUBRT
          ADD   SI,2
LOOP2:    DEC   CX
          JNE   LOOP1
SUBRT:    RET
```

→

```
            ⋮
LOOP1:    MOV   AX,[SI]
          CMP   AX,5
          JB    SUBRT
          CMP   AX,30
          CMC
          JNB   SUBRT
          ADD   SI,2
LOOP2:    LOOP  LOOP1
SUBRT:    RET
```

以来、出る台がさっぱり当たらなくなったでヨ。わしゃ、なんか悪いことでもしたじゃろか……?

パチンコ老人(愛知)

答

最近はパチンコ屋といっても、半分くらいはスロットマシンが置いてあるようです。やはり、パチンコが一番面白かったのは、左手で1つずつ玉を入れながら打つ古き時代じゃなかったでしょうか。

あの頃は、神技のような早打ちができる名人がいたり、ひとつぶの玉を右手で入れては右手ではじくヒマつぶしの老人もいました。それに比べると、現代はスピードの時代。軍資金も玉もアッという間になくなります。

おっと、こんな話は18禁の方には関係のないことですね。あくまでも、問題はプログラムの内容についてです。

修正したプログラムは、オリジナルより1バイト少なくなっています。もちろん、命令を実行する過程はまったく同じです。しかし、このプログラムは実行過程よりも、実行結果に意味があるプログラムなのです。最後のフラグまで見落としてはなりません。LOOP命令はフラグ変化をしないのが特徴です。

まず、このプログラムの目的ですが、SIレジスタで示されるアドレスの中身を、最大でCX回（アドレスを＋2しながら）チェックし、その結果をフラグで返すことです。オリジナルのフラグ結果と、修正後の結果を比較してみましょう。

[SI]を最大CX回チェック	オリジナル	修正後
[SI]＝0～4	CF＝1　ZF＝0	CF＝1　ZF＝0
[SI]＝5～29	CF＝0　ZF＝0	CF＝0　ZF＝0
[SI]＝すべて30以上	CF＝1　ZF＝1	CF＝1　ZF＝不定

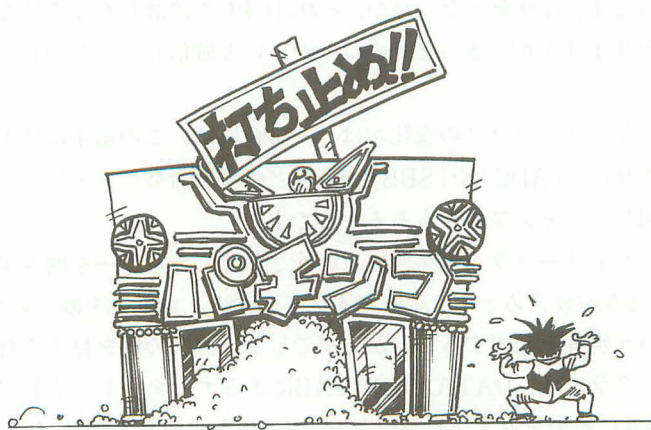
CX回のチェックの間に[SI]＝0～29となれば、フラグ結果はどちらも同じです。しかし、最後まで[SI]が30以上の値であった場合には、修正プログラムではゼロフラグが不定です。不定といっても、ゼロフラグが1になるのは直前の「ADD　SI,2」の演算結果が0となる時だけで、それ以外はすべてゼロフラグは立ちません。

したがって、せっかく最後まで30以上であっても、RET時には途中に0～4があったというフラグ結果になってしまうわけです。これでは、このプログラムの価値も不定になってしまうはずです。

今回は、フラグ変化がないという理由でLOOP命令が使用できませんでした

が、その逆にフラグ変化がないという特徴を利用することもあります。LOOP命令は単にループに便利というだけでなく、このような使い分けをすることも活用したいポイントなのです。

なお、このサブルーチンをコールしたほうのプログラムでは、結果による条件分岐の順序を間違えないように気を付けてください。先ほどのフラグ結果表を見ながら、どの条件で分岐させるべきが確認しておきましょう。それにしても、なんだか危ないプログラムです。

問34 INC命令とキャリーフラグ2

ハロー!! ここはニューヨークです。父の仕事の都合で、半年前にこちらへ家族そろって引っ越してきました。広大な国アメリカは、なんでもビッグです。アイスクリームもステーキも日本の倍はあります。
家では日本語、外では英語、そして夜はマシン語の毎日です。いま、フラグの利用法について色々研究しています。ところで、INC命令やDEC命令がなぜキャリーフラグに対して無変化になっているのかがわかりません。演算結果がキャリーフラグに反映されたほうがプログラムしやすいと思うのですが……。
最近、「＋１」や「－１」する演算の後でキャリーフラグを利用するといったケースにしばしば出会いますが、こんな時には、仕方なく「ADD　reg,1」や「SUB　reg,1」で置き換えてプログラムを組んでいます。

現地名・早ロトム（ニューヨーク）

答

アメリカがなぜ広大かといえば、それが現実だからとしか答えようがありませんね。同じように、INC/DEC命令でキャリーフラグが無変化なのは、それが現実だからです。なぜそうなったか、それはCPUを設計した本人でなければ正確にはわかりませんが、きっとそのほうがいいと確信していたからに違いありません。

確かに、キャリーフラグの変化がほしい場合には、この結果には不満があるかもしれません。「ADC」や「SBB」命令などの演算命令もあるし、キャリーフラグを利用したジャンプ命令もあるからです。

しかし、キャリーフラグが無変化ということは、キャリーを壊さずに「INC」や「DEC」命令が使えるということでもあるのです。無変化を嘆くのではなく、無変化という現実を利用することが先決でしょう。右の例を見てください。

このプログラムは、DATAAとDATABにあるデータを１バイトづつ加算して、DATAAに結果を返すプロシージャですが、もし演算の結果、桁あふれを生じた場合には、次のデータアドレスを求めた後、処理を中断してリターンします。このように、キャリーフラグが無変化であるがゆえに成立するプログラム

```
DATAA   DB      1,2,3,4,5
DATAB   DB      10,9,0FFH,7,6
DTCNT   DW      5

ADDAB   PROC
        MOV     AX,CS
        MOV     DS,AX
        MOV     DI,OFFSET DATAA
        MOV     SI,OFFSET DATAB
        MOV     CX,DTCNT
        CLC
        CLD
ADDLP:  MOV     AL,[SI]
        ADD     [DI],AL
        INC     SI
        INC     DI
        DEC     CX
        JZ      ADRET
        JNC     ADDLP
ADRET:  RET
ADDAB   ENDP
```

もあるのです。実に、先見の明ある合理性ではないですか。この合理性を当然のごとく活用できるようになると、マシン語でも早ロ（早プログラミング？）になれるかもしれませんね。

35 基礎的な疑似乱数

こちらは、第99次南極観測越冬隊の親衛隊です。越冬隊員に憧れて、秘かに南極までやって来ました。昼間はペンギンと戯れ、夜は全員一丸となってパソコンの前に座っています。その目的は、いつ襲ってくるかわからないブリザードを予測するプログラムを組むことです。
しかし、はっきり言ってこれはゲームみたいなものです。というのは、ブリザードを色々と検討した結果、ほとんど乱数に近い出現という結論に達したからです。それはそれでいいのですが、誰もマシン語で乱数を発生させる方法がわかりません。
ブリザードに襲われるのは、まったく乱数というわけでもなく、大きな周期もあるようです。だから、あまり完全無欠の乱数というのもマズイのです。こんな都合のいい乱数なんてあるでしょうか。
アッ、ブリザードに襲われて停電になってしまった……。

ペンギン野郎（南極）

答

南極からの質問……？

どうやら、この質問はモールス信号で送られてきたようです。そもそも、このモールス信号というのは、わからない人にはピッピッピの乱数にしか聞こえません。それでも困った時のために、SOS（トトト ツーツーツー トトト）だけは覚えておくようにと、子供のころよく言われたものです。

例えば、船倉に閉じ込められた時とか、秘密基地に誘拐された時とか……。でも、そんな経験があるはずもなく、いまだに使ったことはありません。

さて、おたずねの乱数ですが、一般的にコンピュータで乱数といっているのは、本物の乱数ではなく疑似乱数が多いのです。これは簡単な計算式によって、一定の周期でランダムに近い数値が得られるようにしたものをいいます。

なんだニセモノか、などとバカにしてはいけません。ゲームなどには、かえってこのほうが適しているのです。というのは、本物の乱数には0が何回も連続するとか、しばらくの間は10以下ばかりとか、本物がゆえに乱数として好ましくない現象が起きることがあるからです。

その点、疑似乱数はいかにも乱数という感じでバラついてくれます。ここに紹介する乱数は、コールするたびにALレジスタにランダムな数値を入れて返すという、最も簡単で一般的な乱数ルーチンです。

```
RND1    PROC
        MOV     AL,CS:SEED1
        MOV     AH,AL
        ADD     AL,AL
        ADD     AL,AL
        ADD     AL,AH
        INC     AL
        MOV     CS:SEED1,AL
        RET
RND1    ENDP

SEED1   DB      0
```

プログラムの内容は、前回の値を5倍し、さらに1を加えたものを新乱数とするというものです。最後に加える数値（ここでは1）は、0以外であれば何でもかまいません。この乱数がどのようにループするのかを示しますので、乱れ方の特徴を確認してください。

```
01 06 1F 9C 0D 42 4B 78 59 BE B7 94 E5 7A 63 F0
B1 76 4F 8C BD B2 7B 68 09 2E E7 84 95 EA 93 E0
61 E6 7F 7C 6D 22 AB 58 B9 9E 17 74 45 5A C3 D0
11 56 AF 6C 1D 92 DB 48 69 0E 47 64 F5 CA F3 C0
C1 C6 DF 5C CD 02 0B 38 19 7E 77 54 A5 3A 23 B0
71 36 0F 4C 7D 72 3B 28 C9 EE A7 44 55 AA 53 A0
21 A6 3F 3C 2D E2 6B 18 79 5E D7 34 05 1A 83 90
D1 16 6F 2C DD 52 9B 08 29 CE 07 24 B5 8A B3 80
81 86 9F 1C 8D C2 CB F8 D9 3E 37 14 65 FA E3 70
31 F6 CF 0C 3D 32 FB E8 89 AE 67 04 15 6A 13 60
E1 66 FF FC ED A2 2B D8 39 1E 97 F4 C5 DA 43 50
91 D6 2F EC 9D 12 5B C8 E9 8E C7 E4 75 4A 73 40
41 46 5F DC 4D 82 8B B8 99 FE F7 D4 25 BA A3 30
F1 B6 8F CC FD F2 BB A8 49 6E 27 C4 D5 2A D3 20
A1 26 BF BC AD 62 EB 98 F9 DE 57 B4 85 9A 03 10
51 96 EF AC 5D D2 1B 88 A9 4E 87 A4 35 0A 33 00
```

また、最後に1を加えないとどうなるか、あるいは5倍ではなく2倍とか3

倍にして1を加えたらどうなるか……。プログラムを少し変更するだけでテストできますから、ぜひ試してみてください。乱数の乱れを改めて認識できるでしょう。

でも、こんなんで本当にブリザードの予測ができるのかどうか、少しばかり疑問が残ってしまいます。

36 相加法による乱数

一生のうち、一度でいいから宝くじの一等に当たってみたい。そう思って20年。もう何枚のハズレくじを買ったことでしょう。当たるのはいつも末等ばかり……。
末等だけは、10枚買えばイヤでも当たります。だから、あれはハズレと一緒です。しかし、ここまでハズレが連続したら、当たる時は一等に違いないはず。それが、唯一の心の支えです。やっぱり私は楽天的でしょうか。
でも、最近は子供も大きくなってきたし、少しはシビアになろうと思い、宝くじの番号を指定して購入するようにしました。もちろん、いつも思い通りの番号が買えるとは限りません。しかし、少なくとも運は向上すると思うのです。ただ、今のところ結果は同じでハズレばかりです。
そこで、今度は購入番号をパソコンの乱数に選ばせようと思います。乱数で各桁ごとの番号を決めたいのですが、私の知っている乱数（５倍して１を足す）では、すぐループしてしまい面白味がありません。もう少し変化に富んだ乱数はできないでしょうか。

宝塚九二男（兵庫）

答

宝くじ。本当に当ててみたいものです。よく「運がよければ……」といいますが、いったいどの程度の運なのでしょうか。

一等の出る確率を計算すると、宝くじによっても、また一等賞金の額によっても違いますが、だいたい百万分の一とか、二百万分の一といった感じが多いようです。いずれにしても、桁が多すぎてピンと来ない数字です。

では、直径１ミリの砂つぶをすき間なく千メートル並べ、その中の１つぶだけが一等になるといったらどうでしょう。……書いてみて、確率の低さに驚いたのは、ほかならぬ自分でした。

さて、例の５倍して１を足すという乱数（問35）ですが、コールするたびに偶数／奇数が入れ替わり、下位４ビットは16回で一巡してしまいます。全体的にも256回で元の値にもどりますから、利用できるのはせいぜい簡単なゲーム程度と考えたほうがよさそうです。

そこで、宝くじに応用できるような複雑な乱数の登場です。といっても、あ

まり計算時間がかからない、相加法というものです。

この基本的な考え方は適当な数値SEED1、SEED2を用意して、コールするたびにSEED1＝SEED2、SEED2＝SEED1＋SEED2を単純に繰り返すものです。

計算は16ビットで行い、その上位8ビットを乱数に取るので、その値からは次回の乱数を予測できなくなります。

```
RND2    PROC
        MOV     AX,CS:SEED1
        MOV     BX,CS:SEED2
        ADD     AX,BX
        MOV     CS:SEED1,BX
        ADD     AX,3711H
        MOV     CS:SEED2,AX
        XCHG    AL,AH
        RET
RND2    ENDP

SEED1   DW      9371H
SEED2   DW      5713H
```

最後に加える 3711_H は特に意味のない数値であり、乱数にさらに乱れを生じさせるためのものです。この数値は、他の初期値とは異なり、0であってもかまいません。また、色々な値を入れてテストしてみてください。

もし、当たりそうな乱数が得られるようだったら、ぜひ教えてもらいたいものです。これは本気です……。

37 乱数利用の基礎

おいでませー、山口へ。山口といえば長州。長州といえば、維新の嵐。革命戦士の心意気でマシン語を制覇するゾ!!

……と意気込んでみたけど、いま挫折しかかっています。ぼくはプロレスラーと精神構造が一体化している高1です。小さい時からバーベルで鍛えてきただけあって、自慢の胸囲は110cmを誇ります。

挫折の原因は単純です。なにかにつけて乱数乱数といいますが、ぼくには乱数の意味はわかっても、それをどう利用するかがわからないのです。例えば、ゲームなどでは敵の動きを乱数で決めることがあるそうですが、動きは8方向程度なのに、乱数では0～255までの数ができてしまいます。

例えば、1～8の乱数がほしければ、その数が出るまで乱数ルーチンをコールしろということでしょうか。

```
                  ⋮
CRND1:   CALL   RND1       ←「RND1」は問35の乱数ルーチン
         CMP    AL,8
         JNB    CRND1
         INC    AL         ←AL=1～8の乱数
                  ⋮
```

あるいは、もっと特殊な乱数ルーチンがあるのですか。どうか、この挫折から立ち直れるよう、カツを入れてください。

男一匹長州軍団（山口）

答

喝（かつ）!!

これで、挫折から立ち直れるなら何度でもカツを入れてあげましょう。喝!!

なに、まだ挫折から立ち直れない……。喝!!

エーッ、まだ……。よく考えたら、こんなことをするより、先へ進んだほうが早く立ち直れそうですね。では、乱数の利用法へと進むことにしましょう。

乱数は、作るよりどう利用するかでその価値が決まります。料理の味がコックさんの腕前で変わるように、乱数もうまく活用することが大切なのです。どんなにいい乱数ルーチンでも、利用方法が悪ければ結果的に悪い乱数になって

しまいます。

例えば、5倍して1を足すという簡単な乱数(問35)を用い、確率1/2の割合で条件分岐させるとします。

A
```
        CALL    RND1
        ROR     AL,1
        JB      PROG1
PROG0:          ⋮
```

B
```
        CALL    RND1
        ROL     AL,1
        JB      PROG1
PROG0:          ⋮
```

AもBもプログラム的には似たようなものです。しかし、得られる結果はまったく違います。Aのほうは、キチンと1回おきに分かれてしまい、乱数を取った意味がなくなっています。一方、Bのほうはある程度予測不可能な分岐をしてくれます。

もちろん、どちらも256回実行すれば分岐割合は同じですが、乱数ルーチンは何もこのプログラムだけで利用されているとは限りません。下手をするとAのプログラムは同じ場所へのジャンプしかしなくなる可能性だってあるわけです。

したがって、簡単な乱数を利用する時には、その乱数の性格も頭に入れて利用することが大切です。「RND2」(問36)を利用すればそういう心配は不要ですが、「RND1」には「RND2」よりプログラムが短い上、速度も速いという特徴があるのです。一概に役に立たないというのではなく、場合による使い分けをしたいものです。

では、質問にあるような数値を乱数で作ってみましょう。乱数ルーチンは「RND2」を使用するものとします。これは、「RND1」では下位4ビットが16回でループしてしまい、今回の応用には適さないからです。

(1) 乱数から0〜7の数を作る

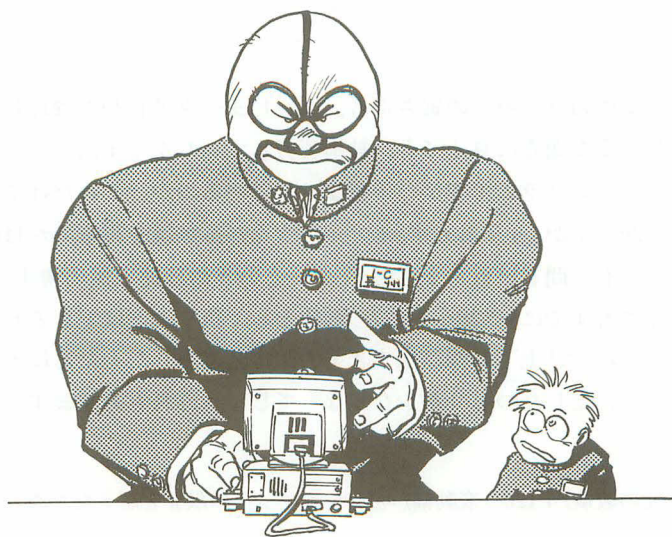
```
CALL   RND2
AND    AL,00000111B
       ⋮
```

(2) 乱数で1〜8の数を作る

```
CALL   RND2
AND    AL,00000111B
INC    AL
       ⋮
```

要するに、乱数ルーチンで得た数値をそのままの姿で利用するか、それとも加工して好みの数値にするか、それを決めるのは「あなた」というわけです。知ってしまえば簡単なものですね。

　挫折から復活の呪文へ……喝（かつ）!!

38 乱数の応用

わたくし生まれも育ちも葛飾柴又です。帝釈天で生湯をつかり、姓は車、名は寅三郎。人呼んで『フーセンの寅』と発します。以後よろしうおたの申します。
わたくしの職業は、全国津々浦々に出向き、立地条件のよい場所を見つけては風船の直営店を開くことであります。そのためには、全国の祭り、縁日、運動会。バザーにイベント、歩行者天国。あらゆる催し物の場所と日時を、正確に記憶しておくことが大切なのでございます。
カバンひとつで長期の出張、つらいけれども気楽な身です。唯一の不安は、行き先を間違えて商売をし損なうことです。しかし、いまでは一枚のフロッピィディスクがあるので安心です。
そこには、柴又のオイちゃんの家でインプットした、全国の催し物の情報が入っております。
こう見えても、プログラムは自前、それもマシン語です。
お祭り好きな日本では、いつもどこかで祭りが開かれています。客足の予想なども、そのプログラムで知ることができます。しかし、欠点はいつも同じ結果しか表示してくれないことです。
そこで、乱数を利用して上下に変化をつけたいと思うのですが、乱数をどのように活用したらいいものか、悩んでおります。どうかよろしくご教授くださいませ。
ちなみに、プログラムはデパートで実行させております。

旅先にて……風船のトラ（福井）

答

地方を巡業中のフーセンの寅さんも、コンピュータで行動を管理していたのですか。気ままな商売に見えても、結構苦労していたんですねェ。

どのようなプログラムで客足の予想までしているのか、これだけではわからないのが残念ですが、とりあえず、＋／－を含んだ乱数値を得るのが目的のようです。もっとも、問2にあるように、＋／－というのは使う側の勝手ですから、何もしなくてもすでに－128～＋127の数値として扱うことはできるわけです。

しかし、これでは上下の幅が広すぎますね。計算をしたとたんに8ビットの限界を超えて、足したつもりが少なくなってしまう恐れもあります。

［例］
150（元の数値）＋120（変動幅）＝14 ←8ビットの限界を超えたため

なかには、客足の予想をするのに、8ビットではそもそも無理があると感じる人もいるでしょう。しかし、数値には単位というものがあります。例えば、100という値でも後ろに「00」を付ければ10000を表現したことになるように、単位次第でいくらでも表現できる幅はふくらむのです。

ゲームのスコアなど、内部では1点単位、画面では飾りの「00」を付けて 100点。そんな見え見えのゴマカシもあるほどです。ということで、「RND2」(問36)を利用して、上下幅のある数値を作ってみます。

(1) −7〜+7の数を作る

```
          CALL  RND2
          AND   AL,00001111B   ←「AND 10000111B」としただけでは
          ROR   AL,1              −7〜+7とはならないことに注意
          JNB   NONEG
          NEG   AL
NONEG:          ⋮
```

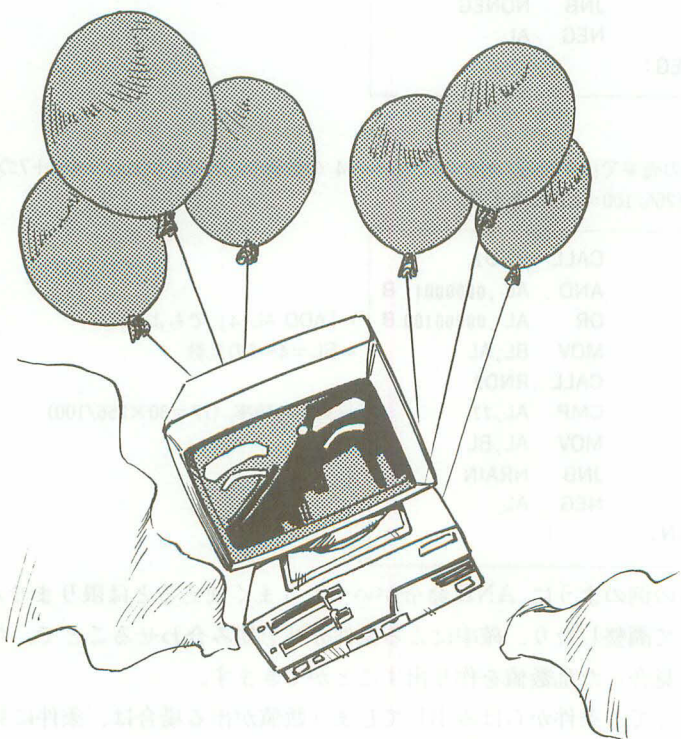
(2) 30%の確率で雨が降る。雨ならば−7〜−4 の数値を、雨でなければ+4〜+7の数値を作る。(256/100=1%と考える)

```
          CALL  RND2
          AND   AL,00000011B
          OR    AL,00000100B   ←「ADD AL,4」でもよい
          MOV   BL,AL          ←BL=4〜7の乱数
          CALL  RND2
          CMP   AL,77          ←30%の確率 (77=30×256/100)
          MOV   AL,BL
          JNB   NRAIN
          NEG   AL
NRAIN:          ⋮
```

これらの例のように AND 命令がいつもうまく使えるとは限りませんが、加減算をして調整したり、確率による条件分岐を組み合わせることで、たいていは条件に見合った乱数値を作り出すことができます。

どうやっても条件からはみ出してしまう数値が出る場合は、条件に見合うま

で同じ処理を繰り返すか、確率の狂いは無視して適当な数値に決めてしまうことです。いい加減でいいのが乱数なのです。

タコはイボイボ、ニワトリゃはだし。イモムシャ19で嫁に行く。乱数いろいろ、よく見りゃ数字。足して引いてりゃ丸くなる……。

39 乱数に変化をつける

わたしは女の子ですが、パソコンが大好きです。とくに、面白いゲームがどういうふうにプログラムされているかを考えると、胸がワクワクします。
ゲームとしては、アクションゲームよりか、ドラクエみたいなRPGが大好きで、わたしもいつの日かあんなゲームを作ってみたいナと思います。だから、少しはマシン語も勉強しました。
敵とか出すのって、乱数を使うんでしょ。それくらいは知っています。でも、ゲームでいう乱数は疑似乱数のはずです。だから、ゲームをスタートして同じように動けば、誰がやっても同じ敵が同じ場所に出てくるはずだと思います。
ところが、ためしにそういう実験を友人とやってみると、ソックリ同じにはなりませんでした。もしかすると、本物の乱数ですか……？

中3あきな（石川）

答

女の子のパソコンファンが近ごろ増えてきているらしいですね。これで、パソコンの未来も明るくなりそうです。

パソコンの魅力の1つに、市販ソフトとまったく同じものを本体で作ることができるという特徴があります。これは、プログラムを組まない人にとっては無用の長物です。その点、遊ぶ面白さから作る面白さに気が付いた石川のあきなちゃんはエライっ!!

未知のものを作る時のスリル、そしてプログラムを初めて実行する時の不安感。これはもう体感RPGそのものです。プラモデルだって、完成してからより、作っている時のほうが楽しいし夢があります。

TVゲームで遊んでいる子供たちが、そういったパソコンの特徴に気が付いた時、パソコンは飛躍的に広がるでしょう。

さて、乱数としては、問36にあった相加法のほかに、コンピュータでは乗算命令を使った乗算合同法に基づく乱数がよく用いられます。これは、単純に前の乱数に259をかけていくものです。なお、この259は、有効ビット数と複雑な乗算合同法の理論により求められるものです。単純に数値をかけるだけですか

ら、プログラムとしては次のように簡単なものです。

```
RND3    PROC
        MOV     AX,CS:SEED1
        MOV     BX,AX         ┐
        SHL     BX,1          │
        ADD     BX,AX         │
        MOV     AH,AL         ├ SEED1×259
        MOV     AL,0          │
        ADD     AX,BX         ┘
        MOV     CS:SEED1,AX
        RET
RND3    ENDP

SEED1   DW      3471H
```

しかし、どんな方法を使うにしても、初期値が同じでプログラムを呼ぶ回数が同じであれば、全く同じ乱数が発生してしまいます。そこで、この乱数の初期値を変えて異なる乱数体系にすれば、ゲームにも変化が生じることになります。

では、どのようにして乱数の初期値を変えたらいいでしょうか。乱数の初期値に乱数を使うというのでは「卵が先かニワトリが先か」になってしまいます。悩みそうな問題ですが、いくつかの簡単な解決法があります。

(1) 名前の合計(文字もマシン語上では数値です)を乱数の初期値にする。

これは RPG などではよくやりますが、名前が同じだと合計も同じになってしまいます。しかし、それも名前による一種のゲーム性です。このプログラムでは、文字コードをアスキーコードで表現し、0を名前のエンドサインにしています。

```
NAMAE   DB      'アキナ',0

START:  MOV     BX,OFFSET   NAMAE
        MOV     DX,0
STAT1:  MOV     AL,[BX]
        OR      AL,AL
```

```
           JZ     STAT3
           ADD    AL,DL
           JNB    STAT2
           INC    DH
STAT2:     MOV    DL,AL
           INC    BX
           JMP    STAT1
STAT3:     MOV    SEED1,DX
                  ⋮
```

（注）DS＝CSと仮定する

(2) キー入力のループの中で、乱数ルーチンをコールする。

一般に、どんなゲームでも最初にタイトルなどがあり、なにかキーが押されるのを待つようになっています。そのキー入力チェックのループで、乱数ルーチンをコールすれば、ゲーム開始時には乱数にズレが生じているというわけです。

(3) コンピュータのタイマーの秒の値を乱数の初期値とする。

コンピュータ本体には、日付や時計機能が備えられているのが普通です。その中で変化が顕著に現れる秒を初期値に利用するものです。

あきなちゃんだけでなく、せいこちゃん、きょうこちゃん、のりこちゃん。それに、さゆりちゃん、わかこちゃん……。みんなパソコンファンになればいいのに……ネ。

40 BPレジスタとは

あなたは神を信じますか？

わたしは神を信じます。わたしはニューギニアにいるドイツ人の牧師です。わたしは神の教えを広め、人々の幸せと平和を祈り、貧しい者、富める者、病める者、健康な者、すべての人々の心の悩みを救うため、ここニューギニアへやって来ました。

ところで、わたしの友人に変な日本人がいます。彼は、わたしに悩みを解決してくれと言います。ところが、その悩みとはなんとマシン語についてです。これには、わたしも困ってしまいます。

彼の話によると、ある日 BP レジスタの存在を知り、いろいろと研究したそうです。その結果、働きとしてはセグメントベースに SS レジスタを使う以外は BX レジスタとほとんど同じような働きをすることがわかったそうです。これは便利と最初のうちは（DS:）でセグメントベースを変更して使っていたそうですが、ふと BP レジスタの存在価値とは何なのだろうと、考えるようになったそうです。

彼は、そのことで悩み、わたしに相談してきました。わたしは人の悩みを解決するのが仕事です。わたしはこの手紙を英語で書き、それを彼に日本語に訳してもらいました。

だから、この手紙はわたしの手紙であっても、書いたのは彼であり、悩んでいるのはわたしであっても、悩みは彼のものであり……。なんで、わたしが悩むのか、それがまた悩みとなって……ナニがどうなっているのか、わからなくなりました。

オー、神よ。悩めるわたしを救いたまえ。アーメン……。

悩める牧師（ニューギニア）

答

「あなたは神を信じますか？」

「信じます。ただし、困ったときだけ」

……というのが、多くの日本人だそうです。無理もないですネ。生まれた時は神社にお宮参り、結婚式は教会で、お葬式はお寺で、そして普段は祈らない……。これが通用するのが日本ですから。

そんないい加減な信者の一人ですが、いつものお礼に悩める牧師さんを救ってあげたいと思います。

まず、BP レジスタを考える前に、あるプロシージャに対してパラメータを渡す方法を考えてみましょう。

1……レジスタを使う方法
2……メモリ変数を使う方法
3……プログラム・コードへ含ませる方法
4……スタックを利用する方法

以上4つの方法が考えられますが、この最後のスタックを用いる方法はしばしばコンパイラなどで用いられているようです。この利点は変数としてのメモリがスタック上にあるために、プロシージャがコールされた時だけ変数が存在する、すなわち、リターンした場合、変数エリアが解放されるため、余計なメモリを消費しないですむのです。そして、このスタック上の変数の参照を前提として生まれたのが、このBPレジスタだったというわけです。

さらに、プロシージャ独自の変数（ローカル変数）を一時的にスタック上にとった場合にも非常に便利な存在となっています。

ですから、1、2、3の方法のみでパラメータ渡しをしていれば、まったくといっていいくらいBPレジスタの存在意義は感じられないでしょう。

では、ここで、パラメータ1つをスタック渡しで、また、初期値、5、13のローカル変数と8バイトのローカルな変数域を持つTEST1というNEARタイプのプロシージャを作ってみましょう。

```
TEST1   PROC    NEAR
        MOV     BP,SP          → SPをBPレジスタへ
        MOV     AX,5
        PUSH    AX             → 初期値5のローカル変数を確保
        MOV     AX,13
        PUSH    AX             → 初期値13のローカル変数を確保
        SUB     SP,8           → 初期値不定のローカルな変数域を8バイト確保
          ⋮
        MOV     CX,[BP+2]      → 親からのパラメータ1の参照例
          ⋮
        MOV     AX,[BP-2]      → 初期値5の変数の参照例
        MOV     BX,[BP-4]      → 初期値13の変数の参照例
          ⋮
        MOV     [BP-6],BX      → 8バイト確保したローカル変数域の使用例
          ⋮
```

```
                  ⋮
           MOV        SP,BP          → スタックを元に戻す
           RET
TEST1      ENDP
```

まず、プログラムの先頭で、BPレジスタへ現在のSPレジスタ値を格納します。次に、各ローカル変数の初期値をAXレジスタを介してスタックへ格納します。さらに、SPレジスタから8を引いて、ローカルな変数域8バイトを確保します。

これで、ローカル変数の確保がすべて終わりましたから、SPレジスタはCALL命令等で間接的に操作はされますが、直接操作することは最後までありません。

ここでBPレジスタの登場となるのです。BPレジスタによるメモリの参照は、SPレジスタと同様に、SSレジスタをセグメントとして暗黙に指定しています。しかしながら、BPはSPとは異なり、CALL命令や割り込み命令等の影響は受けないのです。したがって、安心して変数の参照に使えることになるのです。

さて、プログラムの先頭ではBPにSPを格納していますが、これはSPレジスタを元に戻す時に使えるばかりではなく、「BP＋＊」であれば親からのパラメータであることがわかるし、「BP－＊」であればローカル変数であることがわかるなど、プラス α 的な効果と利点があるのです。

なお、[BP＋0]にはプロシージャ本体の戻り番地が格納されていますからくれぐれも破壊しないようにしてください。また、FARタイプではセグメントアドレスもスタックにキープされていますから注意が必要です。

親からパラメータを渡す場合には、次のように、あらかじめパラメータをスタックへ格納してから呼び出すことになります。

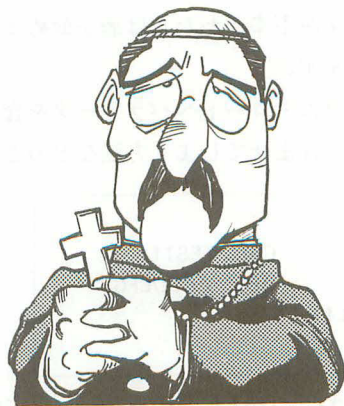
```
          ⋮
MOV       AX,PARA 1
PUSH      AX
CALL      TEST 1
POP       BX
          ⋮
```

プログラムが終了し、「POP ＊＊＊＊」を実行すれば、「＊＊＊＊」へPARA1がプロシージャTEST1で加工された内容を取得することができます(ここではBXレジスタとしている)。

BPレジスタには、このようにとても大切な役割があるのです。とはいえ、汎用レジスタとして用いたり、質問のようにDS:と共に他のメモリ参照にも使えるなど、かなり色々な役割を持たすことができます。ようするに、難しく考えずに自由に使えばよいのです。

これで、悩める牧師さんを救えたとしたら、わたしは神となるのでしょうか。オヤ、本当の神の声が聞こえてきました。

あなたは神ではなく紙。つまり本です、アーメン……。

41 パラメータ渡しプログラム・コード編

ワシは悪魔神官バーボン……。
つまり、酒の好きなおまえたち人間の酔った時の心を司る神じゃ。酒は人の心を気持ちよく狂わせ、理性を失わせる。酒は百薬の長、祝い酒、おとそ、おみき……、飲む理由はいくらでもある。
そこがワシのつけ目じゃ。やがて、ジキルとハイドのように、まったく違った人間ができ上がる。それが、おまえたちの裏の心だ。わかるか、心に表も裏もない。裏が表になれば、それが表になるのだ。その時こそ、世界がワシのものになる。
……というのが、ワシが描く理想の世界なのだが、どうも理想と現実は違うような気がする。いつまでたっても、理想の世界にならんのじゃ。
そこで、ワシは人の心をコンピュータで分析しようと思う。うまい具合いにプロシージャへのパラメータ渡しの方法が問40にあったのでさっそく利用することにしたのだが、第3番目のプログラム・コードへ含ませる方法というのがわからん、そこで具体例と共に解説してほしい、というのが質問じゃ。分析が終わったからといって、ワシの理想通りになるとは限らんから、安心して質問に答えてくれ。

天才バーボン（酒乱界）

答

これは、恐ろしい質問です。私が酒飲みなら、不安でとても答えられないのですが、幸か不幸かわたしは酒が飲めません。だから、平気で質問に答えることができます。

プログラム・コードへパラメータを含ませるとは、次のようにプログラム中にデータを含ませてしまう方法などのことです。

```
          :
        CALL TEST1
        DB    'ABCDEFG$'
LABEL1:
          :
```

このように、CALL文の直後、コード中にパラメータを置くことによって、容易にパラメータ位置を取得することができます。

呼び出されたプロシージャTEST1で注意しなければならないのは、戻り先アドレスがパラメータの先頭アドレスを指しているため、リターンする場合には、この戻り先アドレスを正しい戻り先に更新してから、リターンしなければならないということです。次に例を示しますから参考にしてください。

```
TEST1    PROC
         MOV     BP,SP
         MOV     BX,[BP]
TESL1:   MOV     AL,CS:[BX]
         CMP     AL,'$'
         JE      TSEND
         CALL    PRINT        ←1文字表示ルーチン
         INC     BX
         JMP     TESL1
TSEND:   INC     BX
         MOV     [BP],BX
         RET
TEST1    ENDP
```

ところで、酒を飲むと頭がさえるという人間もいることを、お忘れなく……悪魔神官バーボン殿。

42 ブロック充填

佐渡へ一佐渡へ一と、草木もなびィーく〜……。
と、歌われたのも今は昔のお話です。最近では、佐渡で金が取れたことさえ知らない人がいます。しかし、私はまだまだ佐渡で金は出ると信じています。だからこそ、全財産を処分してここへやって来たのです。
私は、自称地質学者。他称『ヘンなおじさん』です。マ、多少は他称のほうが当たっているかもしれませんが、そんなことはどうでもいいことです。現在、コンピュータにより地層を分析し、夢にまで見た金脈を探しています。
もっとも、コンピュータといっても中古のパソコンです。独学で覚えたマシン語を使っていますが、実行速度が遅く、なかなか分析は進みません。ここにあるのはメモリを一定のデータ(ここでは0)で埋めるルーチンです。

```
        XOR   AX,AX
        MOV   BX,xxxx
        MOV   CX,120
FLOOP:  MOV   [BX],AX
        ADD   BX,2
        LOOP  FLOOP
        RET
```

何度も使われるので速ければ速いほどいいのです。もし、うまく改善してくれたら、金が出た時に1 kgほどあげましょう。

名物・金脈おじさん(佐渡島)

答

1 kgの金……ということは、時価に換算すると……ウ〜ン。いずれにしても大金であることは間違いありません。なんとしても、この質問にだけは答えなければ……。

まず、質問にあるプログラムが何クロックかかっているか、それを計算してみます。その後でどの程度のスピードアップが計れたか、改善したプログラムと実際に比較してみましょう。

ここで注意が必要なのは、メモリに対するアクセスの場合、そのメモリが偶

数番地であるか奇数番地であるかによってクロック数が異なるということです。奇数番地に対するメモリ・アクセスでは、16ビット・オペランドの場合4クロック多く時間を消費します。まず、先ほどのサンプルプログラムを偶数と奇数とに場合分けをして、消費クロック数を求めてみましょう。

```
xxxxが偶数の場合
 3 ×   1 =    3 …… XOR    AX,AX
 4 ×   1 =    4 …… MOV    BX,xxxx
 4 ×   1 =    4 …… MOV    CX,120
10 × 120 = 1200 …… MOV    [BX],AX
 4 × 120 =  480 …… ADD    BX,2
17 × 119 = 2023 …… LOOP   FLOOP（ループ時）
 5 ×   1 =    5 …… LOOP   FLOOP（通過時）
──────────────
    TOTAL：3719クロック

xxxxが奇数の場合
 3 ×   1 =    3 …… XOR    AX,AX
 4 ×   1 =    4 …… MOV    BX,xxxx
 4 ×   1 =    4 …… MOV    CX,120
14 × 120 = 1680 …… MOV    [BX],AX
 4 × 120 =  480 …… ADD    BX,2
17 × 119 = 2023 …… LOOP   FLOOP（ループ時）
 5 ×   1 =    5 …… LOOP   FLOOP（通過時）
──────────────
    TOTAL：4199クロック
```

改善案－①：xxxxが偶数の場合

```
MOV   BX,xxxx      4
XCHG  BX,DI        2
MOV   AX,DS        2
MOV   ES,AX        2
XOR   AX,AX        3
MOV   CX,120       4
CLD                2
REP   STOSW        2+9+10×120=1211
XCHG  BX,DI        2
                   ──────────────
        TOTAL：1232クロック
```

改善案－②：xxxx が奇数の場合

```
MOV    BX,xxxx     4
XCHG   BX,DI       2
MOV    AX,DS       2
MOV    ES,AX       2
XOR    AX,AX       3
MOV    [DI],AL     10
INC    DI          2
MOV    CX,119      4
CLD                2
REP    STOSW       2+9+10×119=1202
MOV    [DI],AL     10
INC    DI          2
XCHG   BX,DI       2
```

TOTAL：1246 クロック

2バイトずつメモリを埋めていたのを、転送命令に置き換えただけですが、実行速度は約7割ほどアップしています。速度を追求する場合、この例では、データが16ビット単位ですから、メモリが偶数か奇数かによって場合分けをしたプログラムを組まなければなりません。そこで、メモリ・ブロックの先頭アドレスを常に偶数番地に取るようにすると、このようなわずらしいことは考えなくても済むようになります。

これで、金1kgが私のものに……なるわけないでしょうね。

43 データ転送

わが輩は、あの有名な「ねずみ小僧次郎吉」の子孫にあたる者だ。もっとも、おまえらだってその子孫の一人かもしれないがな。なにしろ、ご先祖様の子孫は全国にネズミ算式に増えているから、今では誰が子孫かわからなくなってしまったのだ。
だが、わが輩こそ本家本元家元元祖、ただ一人'ねずみ小僧'を名乗れる資格のある子孫なのだ。なぜかって？ 映画で見たご先祖様のご尊顔とソックリだからだ。どうだ、まいっただろう。
ご先祖様は、お金をアルところからナイところへ移動していたが、わが輩が移動するのはお金ではない。メモリにあるデータを、アルところからナイところへ移動しようというのだ。わが輩のプログラムを見てくれ。

```
        MOV   CX,10
MOVEL:  MOV   AX,[BX]
        MOV   [DI],AX
        ADD   BX,2
        ADD   DI,2
        LOOP  MOVEL
          ⋮
```

ご先祖様は、日本一のスピードでお金を移動した。わが輩のプログラムは、その子孫として恥ずかしくないものかどうか、コソッと教えてもらいたい。

根津見狐造（住所不定）

答

ねずみはネズミ算式に増えるといいますが、昔も今もねずみの数はあまり変わらないような気がします。もし、大昔から計算通りに増えていたら、今ごろは世界中がねずみだらけになっていたはずですが、いったいどうなってしまったのでしょう。実に不思議なネズミ算の実体です。

さて、このプログラムではちょっとばかりご先祖様には恥ずかしいようです。ループ命令まで入れると、1ワード転送するのに45クロックもかかっている上、プログラムには速度を追求した形跡がまったくありません。せめて、次のようにするべきです。

```
XCHG    SI,BX
CLD
MOV     AX,DS
MOV     ES,AX
MOV     CX,10
REP     MOVSW
XCHG    SI,BX
        ⋮
```

こうすれば、多少準備に手間がかかりますが、実質1ワードにつき17クロックです。トータルで実行時間は半分以下になっています。もし、メモリが両者共に奇数番地である場合は、次のようにするといいでしょう。

```
XCHG    SI,BX
CLD
MOV     AX,DS
MOV     ES,AX
MOV     CX,9
MOVSB
REP     MOVSW
MOVSB
XCHG    SI,BX
        ⋮
```

なお、MOVSW や MOVSB 命令を REP と組み合わせないで用いた場合、CX レジスタは変化しないということにも注意してください。

では、ご先祖様に負けないよう頑張ってください。

44 データ高速転送

わたしは女の子よ。でも、ちょっと普通の子じゃないの。あまり大きな声じゃ言えないけど、ド・ロ・ボ・ウなの。ア、誤解しないでね。悪人じゃないんだから。
わたしのこと、世間では「怪盗ルビィの指輪」って呼んでいるみたいだけど、普通の人は知らないと思うわ。だって、それは泥棒さんたちの呼び名ですもん。
わたしのお仕事は、泥棒から盗まれたモノを盗り返して持ち主にこっそり返すこと。だから、泥棒さんたちから見れば、わたしが泥棒になるの。もちろん、相手が相手だけにお仕事は楽じゃないわよ。なにしろ相手もプロだから手も足も結構すばやいの。だから、それに負けたらオシマイね。
一番必要なのが頭のすばやさ。パソコンでプログラムを組むときも、高速化は常に考えているわ。もちろん、各メモリブロックの先頭アドレスは偶数番地にしてあるし、データの移動にストリング命令を利用して書くのなんか当然よね。
でも、あの命令ってどちらにしても一方通行でしょ。つまり、SI が＋なら DI も＋、SI が－なら DI も－。わたしがしたいのは、SI が＋で DI が－という転送なの。ここにあるのじゃ平凡よね。

```
        MOV     SI,OFFSET  DATA1
        MOV     DI,OFFSET  RUBII
        MOV     CX,10
XLOOP:  CLD
        LODSW
        STD
        STOSW
        LOOP    XLOOP
        RET
```

やっぱりこれしかないかしら。でも、こんなんじゃ泥棒さんのレベルと一緒よね。きっと、なにかいい方法がありそうな気がするんだけど……。教えて!!

怪盗ルビィの指輪ちゃん（広島）

答

なんか、こうフラフラッと教えたくなるようなお手紙です。きっと、相手の泥棒もすばやさを忘れて、ついボーッとしている間に盗り返されてしまうんでは……。

しかし、プログラムにまで高速性を追求しているとは、まさに本物のプロといった感じです。でも、本当に女の子なんでしょうか。もしかすると、これも敵を欺くための手段かもしれません。

さて、このプログラムを高速化するには、ちょっと頭をヒネらなければいけません。というのは、常識的に考えればこれ以外にプログラムを組む方法がないからです。というわけで、頭をヒネった結果が次のようなプログラムです。

```
        CLI
        MOV     CS:KEEPSS,SS
        MOV     CS:KEEPSP,SP
        MOV     AX,DS
        MOV     SS,AX
        MOV     SP,OFFSET DATA1
        MOV     DI,OFFSET RUBII
        MOV     CX,10
        STD
XLOOP:  POP     AX                      ←8クロック
        STOSW                           ←11クロック
        LOOP    XLOOP
        MOV     SS,CS:KEEPSS
        MOV     SP,CS:KEEPSP
        STI
        RET

KEEPSS  DW      0
KEEPSP  DW      0
```

プログラムの特徴としては、POP命令でデータを1ワードずつ読み出していることです。そのため、LOOP命令を別にして1ワードを転送するのに19クロックで済んでいます。質問にあった方法では、27クロックかかります。

つまり、約3割のスピードアップが実現したわけです。一方、この間はSPを転送するデータアドレスとして使用していますから、当然のことながらレジスタを退避したりサブルーチンをコールしたりすることはできません。もちろん、割り込みも禁止しておかなければなりません。

言ってみれば、これが高速化の代償というわけですが、データが多くなればなるほどこの効果は絶大なものとなるでしょう。また、この転送部分をループを使わずに連記すれば、ループによる無駄が省けさらに高速になります。

せっかくの高速化テクニックですが、こうして本になると泥棒さんにも見られてしまうのでは……？

45 3バイトの加減算

メンソーレ。ぼくはパソコンが趣味の剣道部員です。得意技は、正眼からの鋭いメンです。かけ声は、もちろん「ソーレ!!」です。
ところで、16ビットのレジスタでは0～65535（0～$FFFF_H$）までしか計算できません。いまどき、子供の貯金だってそれ以上です。ただし、ぼくの場合はパソコンを買ったばかりなので、それ以下ですけど……。
でも、ロールプレイングゲームなんかをすると、経験値やゴールドはどんどん上がっていきます。もちろん、どこかに上限はあるでしょうけど、ぼくにはどうやっているのかわかりません。もしかすると、秘密のレジスタでもあるのでしょうか。教えてくれたら、お礼に元気が出る'ハブのエキス'を差し上げましょう。

青い珊瑚礁（沖縄）

答

青い海、澄んだ空、まぶしく輝く太陽。そんな沖縄で見た「ハブとマングースの決闘」は忘れられません。あの時、一匹目のマングースは無残にもハブの毒牙にやられてしまったのです。勝っても負けても、勝負は一瞬で終わりでした……。

さて、問題の16ビット以上の計算方法についてですが、もちろん秘密のレジスタなどあるわけがありません。レジスタがダメなら、頼れるのはメモリだけです。メモリなら何バイト使おうと自由ですからね。例えば、メモリを3バイト使えば、表現できる限界は一挙に 0～16777215 と大幅アップします。

ただし、当然のことながら3バイトを一挙に加減算できる命令などありません。したがって、すべてプログラムで用意することになります。では、3バイト使用した数値どうしの加算と減算のプログラムを組んでみましょう。それぞれの数値はアドレスの若いほうを上位桁とし、計算結果はメモリ（KEKKAからの3バイト）に入ります。

どちらのプログラムにおいても、重要なキーポイントになっているのはキャリーフラグです。また、INC/DEC命令、LODS/STOS命令、LOOP命令では

キャリーフラグが変化しないという特徴が、このプログラムを成立させています。

DATA 0＋DATA 1

```
DATA 0   DB     1FH,20H,5AH
DATA 1   DB     13H,50H,88H
KEKKA    DB     0,0,0

ADD 01   PROC
         MOV    SI,OFFSET DATA0＋2
         MOV    BX,OFFSET DATA1＋2
         MOV    DI,OFFSET KEKKA＋2
         CLC
         MOV    CX,3
         STD
ADDLP:   LODSB
         ADC    AL,[BX]
         STOSB
         DEC    BX
         LOOP   ADDLP
         RET
ADD 01   ENDP
```

DATA 0－DATA 1

```
DATA 0   DB     1FH,20H,5AH
DATA 1   DB     13H,50H,88H
KEKKA    DB     0,0,0

SUB 01   PROC
         MOV    SI,OFFSET DATA0＋2
         MOV    BX,OFFSET DATA1＋2
         MOV    DI,OFFSET KEKKA＋2
         CLC
         MOV    CX,3                  ←CX＝桁数
         STD
SUBLP:   LODSB
         SBB    AL,[BX]
         STOSB
         DEC    BX
         LOOP   SUBLP
         RET
SUB 01   ENDP
```

（注）DS＝ES＝CSと仮定する

これらのプログラムを実用化する場合、BX/SIレジスタを引数としたサブルーチンとすれば、各種の3バイトデータを計算させることができます。

桁数はメモリの許す限り（CXレジスタのカウントできる範囲ですが）増やすことが可能です。もっとも、それを利用する人間が何桁もの十六進数を数値として理解できるかどうかは別問題ですが……。

46 ストリング命令のセグメント・オーバーライド

わらわは卑弥呼……。その昔、わらわはこの耶馬台国を治めておったのじゃが、いまだにわらわが治めた場所がわからぬというではないか。
北九州とか佐賀とか大和地方とか、説は色々と出ているようだが、実はわらわも古いことなのでよく覚えていないのじゃ。というより、わらわの記憶にはインプットされなかったのじゃ。
おっと、わらわは本物の卑弥呼でもなければ、その生まれ変わりでもない。実を申せば、わらわは様々な資料を基に作られたコンピュータ卑弥呼にあるぞ。スイッチポンでわらわは見事に蘇り、そなたに話しかけるのじゃ。
わらわを作ったのは、八丸蜂六（はちまるはちろく）というヒマ人じゃ。わらわが彼と話したところによると、耶馬台国の距離と方角と面積のデータは入れてあるのだそうじゃ。
あとは、データのブロック比較や、ブロックサーチ、ブロック転送などのストリング命令などを駆使して、秘密を解きあかさねばならぬそうじゃが、データが細切れで、しかも色々なセグメントに格納されておるとか……。
なんでも、ストリング命令のセグメント・オーバーライドの方法があればいいらしいのじゃが、わらわを作った時のアセンブラには許されていないというのじゃ。当然のことながら、わらわに分かるわけがないではないか。そこで相談じゃが、ストリング命令のセグメント・オーバーライドの方法を教えてほしいのじゃ。
もしも、この耶馬台国の秘密が解き明かせて、わらわが人間に生まれ変わったなら、そなたの嫁になってやってもよいぞよ。

卑弥呼の部屋（奈良）

答

嫁……だなんて、あまりに恐れ多くて、丁重にご辞退申し上げます。凡人には凡人の歩む道があり、ストリング命令のセグメント・オーバーライドごときで嫁に来られたのでは身体がいくつあっても足りませぬゆえ。

とはいえ、八丸蜂六氏のためにも公開だけはしておきましょう。これは、他力本願であるのか、自力本願であるのかの違いといえます。すなわち、アセンブラがマシン語コードへと変換してくれないのであれば、自分でハンドアセンブルすればよいのです。各セグメント・オーバーライド・プリフィックスのマシン語コードは次のようになります。

```
CS: → 2EH
DS: → 3EH
ES: → 26H
SS: → 36H
```

ですから、このコード値をデータとして、命令コードに含ませてやればよいのです。ここでは、アセンブラの一般的な疑似命令のDBを使ってみました。

```
     ⋮
MOV  SI,OFFSET DATAA
MOV  DI,OFFSET DATAB
MOV  CX,1000H
CLD
DB   2EH        ;2EH=CS:
REP  MOVSW
     ⋮
```

これで強制的にセグメント・オーバーライド（DS→CS）がなされます。ところで、「MOVS」命令にしても、その他のストリング命令にしても、暗黙指定のESを、セグメント・オーバーライドすることはできません。あくまでも、暗黙指定のDSに対してのみ有効となっていますから注意してください。

なお、MASMでは次の書式でセグメント・オーバーライド・プリフィックスが可能となっています。

```
REP  MOVS BYTE  PTR  [DI],BYTE PTR CS:[SI]
REP  LODS BYTE  PTR  CS:[SI]
REP  CMPS BYTE  PTR  CS:[SI],BYTE PTR [DI]
REP  MOVS WORD  PTR  [DI],WORD PTR CS:[SI]
REP  LODS WORD  PTR  CS:[SI]
REP  CMPS WORD  PTR  CS:[SI],WORD PTR [DI]
```

ここでは「DS→CS」というセグメント・オーバーライドを例にとりましたが、上の”CS:”の部分を他のセグメント・オーバーライド・プリフィックス命令に代えることによって任意の組み合せをとることができます。

さて、これでストリング命令のセグメント・オーバライドが可能になったわけですが、例のコンピュータ卑弥呼に利用した後、秘密が解き明かせるかどうかについては一切責任は取れませんので悪しからず……。

問 47 十六進数を十進数に

十数年前、私は海外で番を張っていました。……というと、おそろしい話に聞こえるでしょう。でも、これは本当の話です。
私は、ケンカも弱く嫌いです。でも、旅行は大好きです。ユースホステルでは、よく何連泊もしたものです。
ある時、連泊が1カ月以上になりました。すると、いつの間にか副番長の肩書きが付きました。やがて番長が日本へもどることになり、なんと私が番長へ昇格したのです。そこでは、日本人の最長連泊者を番長とする習わしだったのです。
いま、いい年してパソコンに凝っています。マシン語でも0～9の数字を画面に表示することができるようになりました。それは、AL レジスタに入れた0～9の値を画面に表示するという単純なルーチンです。
しかし、マシン語上で扱う数値は十六進法です。これを十進法に変換し、一桁ずつ AL レジスタに入れられなければ、私のルーチンは役に立ちません。
どうするべきでしょうか。それができないことには、仕事も手につきません。すでに自室へ15連泊もしています。もう、番長は結構だ……!!

番長ボロ屋敷（宮城）

答

知っています、その話。スペインのマドリッドのユースでしょ。あそこでは日本人の番長が代々'火縄式ライター'を安く売ってくれるというので、私もノコノコと出かけて行ったことがあります。大きな日の丸を掲げたベッドの前で、弱そうな番長さんがそれを売ってくれましたっけ。

……もう20年近く前の話です。まだ、伝統の番長さんはいるのでしょうか。どなたか最新情報を教えてください。

さて、十六進数を十進数に直すという作業は、人間とコンピュータとの間を結ぶ重要な仕事です。これがなければ、多くのコンピュータはタダの箱に逆もどりです。そうならないように、プログラマーはそのインターフェイスの役目を果たさなければなりません。

とりあえず、AX レジスタにある十六進数を十進数に変換し、一桁ずつメモリに入れるというプログラムを組んでみます。

```
MEMRY    DB    0,0,0,0,0
HDATA    DW    0FFFFH

HENKAN   PROC
         MOV   AX,HDATA          ←AX＝変換したい十六進数
         MOV   DI,OFFSET MEMRY
         MOV   BX,10000
         CALL  WARIS
         MOV   BX,1000
         CALL  WARIS
         MOV   BX,100
         CALL  WARIS
         MOV   BX,10
         CALL  WARIS
         MOV   [DI],AL
         RET
HENKAN   ENDP

WARIS    PROC
         XOR   DX,DX
         DIV   BX                ←「DX:AX÷BX」を実行し、DIレジ
         MOV   [DI],AL             スタで示されるアドレスにALレジ
         INC   DI                  スタの値を入れる。DIレジスタは
         MOV   AX,DX               ＋1される
         RET
WARIS    ENDP                    (注) DS＝CSと仮定する
```

「HDATA」に十進数に変換したいデータを入れてコールすれば、そのつど「MEMRY」に十進データを得ることができます。このあとで「MEMRY」にあるデータを表示するのは、簡単なことでしょう。あるいは、このサブルーチンにある「WARIS」で、直接ALレジスタの値を表示するようにプログラムを直してもいいでしょう。

マドリッドといえば、マイヨル広場の近くに'森の家'という安くて評判の食堂がありました。貧乏旅行者でも、ここに行けばバカ（肉）がバカバカ食べられると、その名声は遠くおフランスの町まで届いておりましたナ。

いつの日か、また放浪の旅をしたい……。

48 BCD数値の加減算

ワシは海賊船の船長、グルメ・クックだ。
営業海域はカリブ海。ここには無数の宝島があり、欲に飢えた人間どもが金を持って乗り込んでくる。ウワッハッハ……。みな、ワシのお得意様よ。
近ごろでは、金が貯りすぎてとても数えきれん。そこで、コンピュータを使って残高を計算できるようにしようと思うのだ。もちろん、16ビットのレジスタなんかで間に合うわけがない。そういう場合、メモリを使うということも知っておる。
しかし、問題がないわけではない。それを画面にどうやって十進数にして表示するかがわからんのだ。0～$FFFF_H$（0～65535）までの表示法は最近覚えたのだが……。
イヤ、待てよ。できんということもないナ。メモリどうしの引き算（問45参照）は知っているから、百万で割り算をするなら、メモリに$0F4240_H$を入れて引き算を繰り返せばいいのだろう。
でも、面倒くさい話ではないか。いちいち割り算をするなんて。金がドンドン入ってくるのに、こんな面倒なことやってられるかってんだ。
簡単に十進にする方法はないか。オイ、教えろ!!

カリブの海賊（ネズミーランド）

答

カリブの海賊？　まさか、浦安とかロスとかフロリダにいるという例の……。

確かに、あれだけの黄金財宝を数えるとなると、大変なことでしょう。いったい何桁になるのか見当もつきません。割り算のプログラムを組むにしても、割る数が1兆などという桁になると、それ自体を十六進数に変換するのが大変です。

そこで登場するのが、BCD（Binary Coded Decimal＝二進化十進数）による計算方法です。これは、十六進数のうち0～9までを使用し、A～Fは使用しないで計算しようという考え方です。

例えば、これまでは12_Hといえば十進数では18でしたが、これを十進数の12と見なすのです。もちろん、見なすのはあなたですから、計算の相手にA～Fを含んだ数値を使ってしまっては意味がありません。

この条件さえ守れば、プログラム的には簡単です。加算命令のあとに「DAA」、

減算命令のあとに「DAS」とひと言入れてやればいいのです。問 45 の計算例と比べてみてください。

DATA 0＋DATA 1

```
DATA 0  DB     32 H,20 H,58 H
DATA 1  DB     13 H,50 H,88 H
KEKKA   DB     0,0,0

ADD 01  PROC
        PUSH   DS
        PUSH   ES
        MOV    AX,CS
        MOV    DS,AX
        MOV    ES,AX
        MOV    SI,OFFSET DATA0＋2
        MOV    BX,OFFSET DATA1＋2
        MOV    DI,OFFSET KEKKA＋2
        CLC
        STD
        MOV    CX,3
ADDLP:  LODSB
        ADC    AL,[BX]
        DAA
        STOSB
        DEC    BX
        LOOP   ADDLP
        POP    ES
        POP    DS
        RET
ADD 01  ENDP
```

DATA 0－DATA 1

```
DATA 0  DB     32 H,20 H,58 H
DATA 1  DB     13 H,50 H,88 H
KEKKA   DB     0,0,0

SUB 01  PROC
        PUSH   DS
        PUSH   ES
        MOV    AX,CS
        MOV    DS,AX
        MOV    ES,AX
        MOV    SI,OFFSET DATA0＋2
        MOV    BX,OFFSET DATA1＋2
        MOV    DI,OFFSET KEKKA＋2
        CLC
        STD
        MOV    CX,3                     ←CX＝桁数
SUBLP:  LODSB
        SBB    AL,[BX]
        DAS
        STOSB
        DEC    BX
        LOOP   SUBLP
        POP    ES
        POP    DS
        RET
SUB 01  ENDP
```

なお、「DAA」、「DAS」、共に演算結果は AL レジスタに求めるという条件付きですから注意してください。

計算の終了した時点で、「KEKKA」にはそのまま十進数として読めるような形で答が入っています。画面に表示する時には、それぞれの値を上位／下位に分けて表示すればいいのです。

上位の値を AL レジスタへ

```
MOV   AL,[BX]
MOV   CL,4
SHR   AL,CL
```

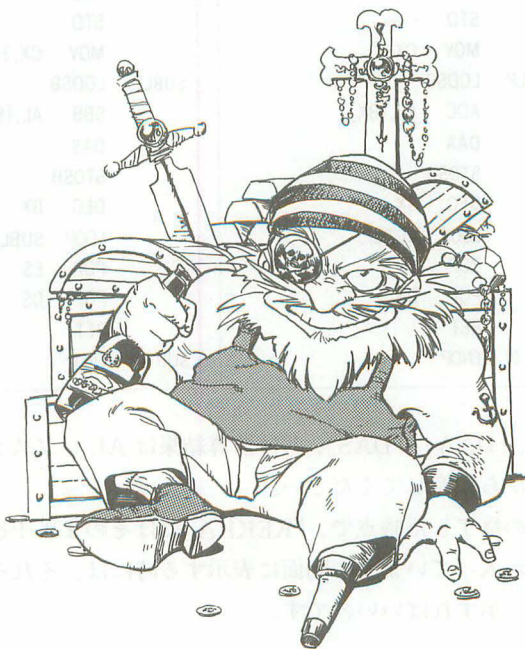
下位の値を AL レジスタへ

```
MOV   AL,[BX]
AND   AL,00001111B
```

（注）どちらも DS:BX レジスタで表示したい数値のあるアドレスを指定したものとする

これで、割り算による手間が省けるとともに、桁（兆の位とか億の位…等）を意識しないで数値を表示することができるわけです。

ところで、カリブの海賊は有名だけど、ネズミーランドっていうのは？？？

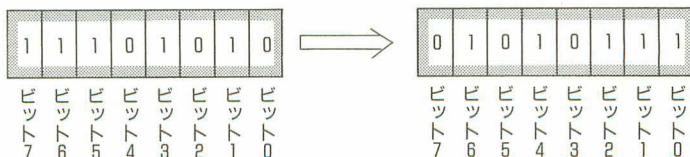
49 データの左右反転

初めての質問です。雑誌に投稿するのが趣味の高校3年ですが、これからは受験勉強のため投稿はしばらく休まなければなりません。でも、これまでに某誌に2回も自分の作品が発表されたことがあるんですよ（BASIC＋マシン語だよ〜ン）。
マシン語は、おもにキャラクタの表示部分なんだけど、左右反転したパターンを見るたびに、メモリがもったいないなァと思います。
インストラクション表で、左右反転ができるような命令を捜してみましたが、どうもなさそうです。
なんとか、この無駄を解決する方法はありませんか。できれば、受験の前にこのモヤモヤを取り除きたいなァ。

投稿マニア（徳島）

答

さすが、投稿マニアを自称するだけあって、鋭い質問ですね。データは少なく、プログラムは短く、そして速く……マシン語の技術の差は、ここに現れるのです。

すでに、このことに目覚めているからこそ、これだけの質問がでるのでしょう。おそらく、受験勉強がなければ自力で解決できるだけの実力があるはずです。

とりあえず、質問の内容が具体的でないので、例をあげて解決することにします。

左右反転のパターンを作るには、1つのデータを図のように左右対象に入れ換えなければなりません。残念ながら、この命令はいくらインストラクション表を捜してもありません。また、論理演算命令を繰り返しても作ることはできません。

ということは、プログラムで反転データを作るしかないわけです。そこで、キャリーフラグを利用して次のようにします。

```
        MOV   CX,8
LP01:   RCL   AL,1
        RCR   AH,1
        LOOP  LP01
        MOV   AL,AH
```

ALレジスタの値を左回りにローテートさせると、7ビット目の値がキャリーフラグに入ります。次に、キャリーフラグの値がAHレジスタの7ビット目に入るようにAHレジスタを右にローテートさせます。これを8回繰り返すわけです。

キャリーフラグの動きは、インストラクション表を見ればわかるでしょう。問題は少々時間がかかることで、速度を追求する場合には利用できないかもしれません。それを考慮した上で使えば、結構役に立てられるはずです。

でも、この程度の内容の命令は最初から用意しておいて欲しいものですネ。

50 実用できる左右反転プログラム

P。これは「ピー」と読みまP。こちらは、ミニ独立国家ピータンでP。わが国の憲法はただひとつ、言葉の最後にPを付けることでP。

では、質問でP。最近、データを反転するテクニック（問49参照）を覚えたのですが、どうも実用性に乏しいような気がするのでP。その理由は速度でP。1バイトのデータを反転するのに、

```
         MOV    CX,8      ←4
LP01:    RCL    AL,1      ←2×8=16
         RCR    AH,1      ←2×8=16
         LOOP   LP01      ←17×7+5=124
         MOV    AL,AH     ←2
```

で、トータル162クロックもかかっていまP。たった1バイトならともかく、大量のデータを反転するとなると問題でP。しかも、そのためにCXレジスタを破壊しなければならないのでP。

これでは、現実には使いたくても使えないということになってしまいまP。もっとうまい方法があると、とても便利でハッピーな気分になれそうでP。

ピータン国王（岩手）

答

さすが、国王。単にプログラム・テクニックを知っただけでは満足せず、そこに実用性を求めていまP。おそれ入りましたでP。

確かに、1バイトを反転するのに162クロックもかかるのは問題でしょう。グラフィックなどはデータの山ですから、リアルタイム・ゲームに応用するのはほとんど不可能とも言えます。

そこで、少々メモリは必要ですが、とっておきの秘法を紹介します。

この場合、DSレジスタには、RDATAの存在するセグメント値が、あらかじめ設定してあるものとします。

これで、実行速度は1バイトにつき15クロックですから、147クロックも速くなったことになります。また、ALレジスタを反転するのにCXレジスタを必

```
RDATA LABEL BYTE
        DB      00H,80H,40H,0C0H,20H,0A0H,60H,0E0H
        DB      10H,90H,50H,0D0H,30H,0B0H,70H,0F0H
        DB      08H,88H,48H,0C8H,28H,0A8H,68H,0E8H
        DB      18H,98H,58H,0D8H,38H,0B8H,78H,0F8H
        DB      04H,84H,44H,0C4H,24H,0A4H,64H,0E4H
        DB      14H,94H,54H,0D4H,34H,0B4H,74H,0F4H
        DB      0CH,8CH,4CH,0CCH,2CH,0ACH,6CH,0ECH
        DB      1CH,9CH,5CH,0DCH,3CH,0BCH,7CH,0FCH
        DB      02H,82H,42H,0C2H,22H,0A2H,62H,0E2H
        DB      12H,92H,52H,0D2H,32H,0B2H,72H,0F2H
        DB      0AH,8AH,4AH,0CAH,2AH,0AAH,6AH,0EAH
        DB      1AH,9AH,5AH,0DAH,3AH,0BAH,7AH,0FAH
        DB      06H,86H,46H,0C6H,26H,0A6H,66H,0E6H
        DB      16H,96H,56H,0D6H,36H,0B6H,76H,0F6H
        DB      0EH,8EH,4EH,0CEH,2EH,0AEH,6EH,0EEH
        DB      1EH,9EH,5EH,0DEH,3EH,0BEH,7EH,0FEH
        DB      01H,81H,41H,0C1H,21H,0A1H,61H,0E1H
        DB      11H,91H,51H,0D1H,31H,0B1H,71H,0F1H
        DB      09H,89H,49H,0C9H,29H,0A9H,69H,0E9H
        DB      19H,99H,59H,0D9H,39H,0B9H,79H,0F9H
        DB      05H,85H,45H,0C5H,25H,0A5H,65H,0E5H
        DB      15H,95H,55H,0D5H,35H,0B5H,75H,0F5H
        DB      0DH,8DH,4DH,0CDH,2DH,0ADH,6DH,0EDH
        DB      1DH,9DH,5DH,0DDH,3DH,0BDH,7DH,0FDH
        DB      03H,83H,43H,0C3H,23H,0A3H,63H,0E3H
        DB      13H,93H,53H,0D3H,33H,0B3H,73H,0F3H
        DB      0BH,8BH,4BH,0CBH,2BH,0ABH,6BH,0EBH
        DB      1BH,9BH,5BH,0DBH,3BH,0BBH,7BH,0FBH
        DB      07H,87H,47H,0C7H,27H,0A7H,67H,0E7H
        DB      17H,97H,57H,0D7H,37H,0B7H,77H,0F7H
        DB      0FH,8FH,4FH,0CFH,2FH,0AFH,6FH,0EFH
        DB      1FH,9FH,5FH,0DFH,3FH,0BFH,7FH,0FFH

                ⋮
        MOV     BX,OFFSET RDATA
        XLAT
                ⋮
```

(注) XLAT :AL ← DS :[BX＋AL]

要としませんので、ループの中にこのまま組み入れることも可能です。また、ループ中に組み込む場合、BXレジスタの初期値をループに入る前に設定すれば、1バイトにつき11クロックですから、さらに変換スピードはアップします。

アイデアがシンプルなだけ、実用性は逆に高くなったと言えるでしょう。大いに活用してください。

では、国王。バイP……。

51 AFフラグとは

グ　ワッハッハ……!!
オレ様は地獄の番人、閻魔大王だ。グワッハッハ……!!
オレ様の前でウソなどついてみろ。アッという間に、この大きな‘やっとこ’で舌を抜いてしまうからナ。オレ様は、人間の話す言葉から、それがウソか真実か、簡単に見抜くことができるのだ。
この間も、「手みやげに金の延べ板を持参してきました」と、オレ様に鉛に金メッキをしたまがい物を渡そうとしたヤツがおってな。もちろん、血の池経由の針山行きよ。バカ者めが……。
ところが、このオレ様にも最近弱点ができてしまったのだ。それは、なんとマシン語を話題にする相手のことだ。オレ様も必死になってマシン語を勉強したのだが、なかなか思うように上達せんのだよ。
今日も、「フラグの中にはAFフラグなど、ユーザーには関係のないフラグがある」などと言われてしまい、どうにもわからんで困っておる。なんとか、オレ様の役目も理解して、親切にご指導願いたい。

閻魔大王クッパ（地獄一丁目）

答

こわそーッ。でも、ここでウソなど教えたら、いつの日かお目にかからなければならなくなった時、血の池経由の針山行きなどとされてしまいそうです。それだけは、なんとしても避けなければ……。

とりあえず、AFフラグの内容とは次のようなものです。

・AF（補助キャリー）フラグ
演算の結果、その演算がバイト単位であればニブル、ワード単位ならばバイトでの桁上がり、または桁借りが生じた時にセットされる。

このフラグは、BCD演算や十進ASCII演算における演算補正のために用意されているものです。したがって、これらの演算に関係のない命令によって変化しても、基本的に意味はありません。ユーザーがジャンプ命令等で利用できないようになっているのも、特に必要がないからだといえるでしょう。

しかし、DAA命令によっても変化するAFフラグは、利用できるなら利用し

たい時がないわけでもありません。例えば、特定の桁が変化したかどうかを調べるという場合など、AF フラグがキャリーフラグとともに利用できると便利です。

このような場合でも、AF フラグがフラグレジスタの中にある限り、どうやっても調べることはできません。そこで、AF フラグの存在するフラグレジスタの下位 8 ビットを、次のように AH レジスタに移動すれば調べられます。

```
LAHF                ←フラグレジスタの下位8ビットをAHレジスタへ
TEST    AH,10H      ←AFフラグのチェック
JNZ     xxxx
         ⋮
```

また、あまり関心しませんが次のような方法も考えられます。

```
PUSHF
POP     AX          ←フラグレジスタの値をAXレジスタへ
TEST    AL,10H      ←AFフラグのチェック
JNZ     xxxx
         ⋮
```

この方法であれば、もちろん上位 8 ビットのフラグも調べることはできますが、いずれにしてもそれほど必要になることはないでしょう。要するに、やむを得ない時には、こういった奥の手もあると覚えておけばいいという程度のことです。

これらは、決してウソの説明ではありません。本当に、本当ですとも……閻魔大王クッパさま……。

52 「CMP DS,ES」を実現

拝啓

春光うららかな今日このごろ、皆様にはますますご隆盛大慶に存じます。

わが輩は、明治生まれの真空管技術者でござりまする。わが輩も、一時期は時代の最先端をいく花形技術者としてもてはやされたのでござりまするが、それも今や過去の夢物語であります。

そこで一念発起、パソコンを購入しマシン語なるものにチャレンジしたのでござりまする。結構、色々な命令があるものでござりますな。これまでの調べで、他のレジスタに比べて、セグメント・レジスタを操作する命令が少ないことも判明しております。

セグメント・レジスタである以上、ある程度はやむを得ないのかもしれぬが、それにしてもセグメント・レジスタ用のCMP命令がないのはどういうことでござろうか……。

つまり、「CMP CX,DS」とか「CMP DS,ES」という命令が欲しいのでござりまする。いくらセグメント・レジスタとはいえ、この程度の能力がないようでは、なにかと不便で仕方がないのでござりまするな。

よろしくお願もうすでござる。

敬具

明治キメラ（長野）

答

これはこれは、ご丁寧なご質問ありがとうでござりまする。パソコンという共通の話題がある限り、「明治は遠くなりにけり」なんて死語でござりまするな。

確かに、セグメント・レジスタ用のCMP命令がないというのは不便なことかもしれません。例えば、セグメント・レジスタをワーク・レジスタとして使う、なんていう場合もありますから。

しかし、命令として存在しないものは自力で作ればいいのです。そもそも大した命令がないのがマシン語ですから、ないものはプログラムで作るのが当然ともいえます。さっそく、質問にある2つの命令を実現できるように、マクロ定義をしてみました。

```
CMPR1R2  MACRO  REG1,REG2
         PUSH   AX
         PUSH   BX
         MOV    AX,REG1
         MOV    BX,REG2
         CMP    AX,BX
         POP    BX
         POP    AX
         ENDM
```

このようにマクロ定義すれば、ほとんどのレジスタが使えますから、いつでもセグメント・レジスタのCMP命令として利用することができます。質問を例にとって、ここで定義したマクロの使用方法を次に示します。

(CMP CX,DS) の場合

```
    ⋮
CMPR1R2   CX,DS
JGE       ****
    ⋮
```

(CMP DS,ES) の場合

```
    ⋮
CMPR1R2   DS,ES
JNE       ****
    ⋮
```

この例ではCMP命令でしたが、CMP命令に限らず、他の命令、例えばADDとかSUBなどをマクロ定義してみてはいかがでしょうか。では、がんばってくださいでござりまする……。明治キメラ殿。

53 「CMP DX:AX,＊＊＊」を実現

京都～ォ、大原三千院……。
うちは京の舞妓どす。優雅に歌など歌っている姿を想像して、うちの美しさを味わっておくれやす。
でも、これはマイコンが趣味でドス（DOS＝Disk Operating System）を作りたいという気持ちが高まって、いつの間にか自分のことを舞妓はんと思うようになったのが本当の話どす。
勘違いされる前に宣言しておきやすけど、一応正真正銘の女の子どすよってに、気色ワル～イとかは言わはらんようにお願いしやす。ちなみに、うちの話す舞妓はん言葉もメチャクチャどすェ。なにしろ、生まれは関東どすから……。
いま、あるプログラムで、DXに上位16ビット、AXに下位16ビットを格納した、32ビットの数値とイミーディエイト値の比較「CMP　DX:AX,12345678H」という内容を実行したいのどすが、うちの実力ではどうにもわからんのどす。
こういう場合、いったいどうしたらいいのどす？　困ってしまうどす。ア～ァ、それにしてもマイコン以上に舞妓はん言葉は難しいどすねェ。

ニセ舞妓（京都）

答

言葉の最後に「どす」を付ければ舞妓さんというのも、なにかモノスゴイ勘違いのような気がしますどす。とりあえずは、正真正銘の女の子ということが唯一の救いかもしれませんどす。

さて、この命令に相当するプログラムは、次のようにすれば作ることができます。

CMP命令の動作は、第一オペランドから第二オペランドを減算し、その計算結果をフラグへ反映させるものです。動作としては、第一オペランドの値が命令実行後も元のままであること以外はSUB命令と同じです。したがって、レジスタを保存してやれば、このように32ビットCMP命令が簡単に作れることになるのです。

ここではプロシージャとして記述しましたが、マクロ定義をしておくのも1つの方法でしょう。

```
CMPDA   PROC
        PUSH  AX
        PUSH  DX
        SUB   AX,CS:DATA
        SBB   DX,CS:DATA+2
        POP   DX
        POP   AX
        RET
CMPDA   ENDP

DATA    DW    5678H
        DW    1234H
```

54 レジスタペアのNEG命令

ここはエジンバラ……。イギリス北部、スコットランドにある美しい町です。申し遅れましたが、ボクはイギリス国内をヒッチハイクによって旅している者です。
慣れてしまえばどうということのないヒッチハイクも、最初は親指を立てて右手を上げるのにとても勇気が入りました。一台、また一台……と車が通過するたびに、ボクには絶対に止まってくれないのではないかという不安感が頭をかすめたものです。
1時間ほどして、ようやく一台の乗用車が止まってくれた時は、ボーッとして一瞬われを忘れたほどです。でも、それがきっかけとなってヒッチハイクのコツがわかり、今ではマナーをわきまえた一流のヒッチハイカーです。
それはいいとして、忘れられないのが日本に置いてきたコンピュータのこと。というより、せっかく覚えたマシン語のことです。だから、頭の中はいつもマシン語でいっぱいです。車が止まってくれない時は、もちろんマシン語のことを考えています。
「NEG」というマイナスの値を作る命令がありますね。ある時、あれの32ビット版はどうやったらいいか。これを考えていたら、右手を上げるのをすっかり忘れてしまったことがあります。
おかげで、車は止まらないわ、雨は降ってくるわ……で、さんざんな目にあってしまいました。どうか、気分よくヒッチハイクができるように教えてください。

ヒッチくん（エジンバラ）

答

日本ではヒッチハイクの習慣はあまりありませんが、ヨーロッパなどでは若者の移動手段として立派に市民権を得ています。目的地にいつ到着するかわからないという不確実な点も、若者の気ままな旅に合っているのかもしれません。

ヒッチハイクのコツは、まず郊外の街道筋まで出るということ。例えば、銀座や新宿のような場所では成功しません。そして、車が止まってくれたら、急いで車のほうへ走り寄り、行き先をハッキリと言います。行き先が違えば、残念ながら乗るのはあきらめなければなりません。運よく乗せてもらうことができたなら、相手に不快感や不安感を与えないようにするのも大切なことです。乗せて楽しかったと思われるようになれば、もう一流のヒッチハイカーでしょう。

私は五円玉をたくさん用意しておいて、降りる時に記念に一枚渡しました。外国には穴の開いたコインが少ないので、結構喜ばれたものです。おっと、あまりヒッチハイクの秘伝ばかり書いていると、肝心の質問を忘れてしまいそうなので、ここではこれくらいにしておきましょう。

16ビットの「NEG」は問14で紹介しましたが、32ビット用の「NEG」は命令として存在していません。そこで、プログラムで作ることになります。データは上位16ビットがDXへ、下位16ビットがAXへあるものとします。

```
NEG32:  SUB AX,1
        SBB DX,0
NOT32:  NOT  AX
        NOT  DX
        RET
```

初めに、「SUB AX,1」としていることに注意してください。これを「DEC AX」と置き換えてしまうと、とんでもないことになります。というのも、「DEC」や「INC」命令では、キャリー・フラグが変化しないため、次の「SBB DX,0」という命令が無意味になってしまうからです。

また、プログラムのラベルを見ておわかりのように、最初の減算命令を省くことにより「NOT」の32ビット版とすることができます。

ちなみに、ビット単位で「NOT」と「NEG」の違いを見ると、次のようになります。ALレジスタの値は 00001111B=$0F_H$と仮定します。

NOT AL: 11110000 B = $F0_H$ (数値上は「XOR AL,0FFH」と同じだが、フラグ変化が違う)

NEG AL: 11110001 B = $F1_H$ (NOTした値に＋1したもの)

両者とも、内容をよく理解した上で使用することが大切です。フラグ変化の違いなどもよく確認しておきましょう。気分よくヒッチハイクするためにも……。

55 ジャンプ代わりにリターン

先手５八玉……。
どうだ!! これがワシの発明した戦法「玉先導」の一手目だ。その先はまだ秘密で教えられないが、この手で動揺した棋士は皆あわてふためいて、結局は負けてしまうことになるのだ。ワッハッハッハ!!
ワシはプロの棋士、相手もプロ……。といっても、相手のプロはプログラムで動くコンピュータという意味だがな。もちろん、ワシがプロの棋士というのも、プログラムで動く棋士という意味だ。
おっと、正確にはワシはそのプログラマーというわけだ。できるだけ強いプログラムにするには、メモリを効率よく使って思考ルーチンを強化しなければならぬ。実は、プログラムの一部に50箇所にジャンプする部分がある。当然、テーブルは使っている。

```
TABLE     DW     PRO00,PRO01,……,PRO48,PRO49

MAINP:              ⋮
          MOV    BX,OFFSET TABLE
          SHL    AX,1
          ADD    BX,AX
          JMP    CS:[BX]
```

プログラムの概略は以上なのだが、それぞれのルーチンからは「MAINP」へジャンプしてもどってくるようになっておるのだ。単純に計算しても「JMP　MAINP」が50個はあるということだ。そのほかにも、条件分岐でもどるものもあるので、実際には100箇所以上から「MAINP」へジャンプしてくるのだ。
こういったルーチンがいくつかあるので、これに要するメモリもバカにならない。なんとか省メモリ化できないだろうか。

白夜の棋士（静岡）

答

コンピュータで本当に強い将棋プログラムができたら、つまり本物のプロ棋士と互角に戦えるようなレベルになったら、こんな痛快なことはないでしょうね。

なぜかというと、そこまで強いプログラムになると、理論上は常に最善手を

指してくるからです。そうすると、敗着の手を指す恐れのある人間の頭では、アッという間にコンピュータに勝てなくなってしまうでしょう。では、そんなコンピュータどうしの勝負はどうなるのか、それを早く見てみたいものです。

しかし、現在のコンピュータのレベルでは、大局観という要素があまりないので、当分はお遊びソフトの域を出るのは難しそうです。

さて、質問にあるようなメインプログラムへの大量ジャンプですが、こういったケースにはアチコチで遭遇します。省メモリを目指す場合には、次のようなテクニックを覚えておくと便利です。

```
TABLE      DW     PRO00,PRO01,……,PRO48,PRO49

MAINP:            :
           MOV    BX,OFFSET MAINP         ←MAINPのアドレスを、
           PUSH   BX                        もどりアドレスとして
           MOV    BX,OFFSET TABLE           スタックへ入れておく
           SHL    AX,1
           ADD    BX,AX
           JMP    CS:[BX]
```

つまり、「MAINP」をリターンアドレスとしてスタックへ退避しておくのです。こうすれば、それぞれのルーチンからJMP命令でなくRET命令で「MAINP」へジャンプさせることができます。

この時注意しなければならないのは、各ルーチン先で場合によっては「MAINP」へもどらないことがある場合です。このような場合、スタックへリターンアドレスを退避してあることを忘れてしまうと、SPが狂って暴走する危険性があります。したがって、RET命令で直接「MAINP」へもどらない時は、SP合わせのダミーとして「POP　AX」などを実行するか、SP+2としてSPを元にもどす必要があります。

このことにさえ注意すれば、このテクニックの活用の場は結構多いはずです。うまく活用して、本当に名人級の将棋ソフトを作ってください……。

56 ビット情報の縦方向コピー

ホワァ～ァ……。眠たい眠たい……。
プログラマーは眠たい。とにかく眠たい。おっと、これは失礼。私は、某ソフトハウスでゲーム制作のプログラム・アシスタントをしている者です。
アシスタントといっても、自分の受け持ったルーチンは責任を持って組まなければなりませんから、手伝いというような甘い考えは通用しません。しかし、恥ずかしながら私はプログラマーとしてはまだまだ未熟者です。
ところで、私が今やろうとしているのは、AL レジスタのビット 0 から 7 までのビット情報を、メモリのビット N へ80バイトごとにコピーするという作業です。つまり、横方向のビット情報を縦方向にコピーすることなのですが。どうも、プログラムまで寝たまんまです。どうしたらいいでしょうか？
それにしても、ホワァ～ァ……眠たい眠たい……。

眠りプログラマー（福島）

答

ホワァ～ァ……眠たいのは、こちらも同じこと。まったく、いくら寝ても眠たさは解消しません。今も、必死に眠たさをこらえて質問に答えているところです……。

それでは、こちらの眠たさはこらえて、この横に寝たまんまのプログラムを、起こして縦にしてみせましょう？？？

実は、これはビット情報を縦方向へコピーする基本形なのです。これを応用すると、例えばグラフィック VRAM 上に展開した横向きの戦闘機を、縦に向きを変えて表示するという芸当ができるわけです。もっとも、高速処理が要求されるリアルタイムゲームでは、あらかじめデータとして持たせなければなりませんが……。

ホワァ～ァ……。本当に眠たくなってしまった、ムニャ……。

```
ADR01   DD      ASEG1 : AREA1
BITN    DB      7

AWAKE   PROC
        PUSH    DS
        MOV     DX,0FE01H
        MOV     CL,CS : BITN
        ROL     DH,CL
        ROL     DL,CL
        LDS     SI , CS :ADR01
        MOV     CX,8
AWAL1 : AND     [SI],DH
        SHR     AL,1
        JNB     AWA01
        OR      [SI],DL
AWA01 : ADD     SI,80
        LOOP    AWAL1
        POP     DS
        RET
AWAKE   ENDP
```

57 ブロック比較について

ゲ、ゲ、ゲゲゲのゲーッ……。
近ごろ、ウイルスというコンピュータを悩ます妖怪がはびこっているそうだが、そんなことで困っている人は、この正義のコンピュータ妖怪「ゲゲゲのQ太郎」に連絡してください。連絡方法は、お近くのゲゲゲ・ネットワークへ申し込めばいいようになっています。あなたのシステムに入り込んで、即座にウイルス妖怪をやっつけてあげましょう。
……という看板を出したのはいいけれど、実はまだ本物のウイルス妖怪にはお目にかかったことがありません。つまり、この看板は「出たとこ勝負」のバクチなのです。とはいえ、とりあえずは正体不明の妖怪対策も考えておかなければなりません。
たぶん、ウイルス妖怪退治には値をサーチする命令や、ブロック比較命令が欠かせないでしょう。これらの命令としては、次のようなものがありますが……。

SCASB ：AL と ES:[DI]とを比較し、DF=0なら DI ← DI+1、DF=1なら DI ← DI−1を実行する。REP (REPZ) または REPNZ 命令と組み合わせると、CX レジスタをカウンターとした連続比較が可能となる。

SCASW ：AX と ES:[DI]とをワード単位で比較し、DF=0なら DI ← DI+2、DF=1なら DI ← DI−2を実行する。REP (REPZ) または REPNZ 命令と組み合わせると、CX レジスタをカウンターとした連続比較が可能となる。

CMPSB ：DS:[SI]と ES:[DI]とをバイト単位で比較し、DF=0なら DI ← DI+1、SI ← SI+1を、DF=1なら DI ← DI−1、SI ← SI−1を実行する。REP (REPZ) または REPNZ 命令と組み合わせると、CX レジスタをカウンターとした連続比較が可能となる。

CMPSW ：DS:[SI]と ES:[DI]とをワード単位で比較し、DF=0なら DI ← DI+2、SI ← SI+2を、DF=1なら DI ← DI−2、SI ← SI−2を実行する。REP (REPZ) または REPNZ 命令と組み合わせると、CX レジスタをカウンターとした連続比較が可能となる。

(注) DF：ディレクション・フラグ

実は、どうも実行結果やフラグの見方がわからないのです。コンピュータの正義のために、正しい利用法を教えてください。

ゲゲゲのQ太郎（正マ界）

ゲゲゲのQ太郎……？　どこかで聞いたような名前です。頼りになる妖怪と

頼りにならないお化けが合体したような、なんとも妙な雰囲気が漂っていますが、質問からするとまだまだ実力不明の妖怪のようです。

W、Bとも基本的な考え方は変わりありませんから、ここではバイト単位の命令について説明をします。まずSCASBですが、これを実行するとフラグ以外のレジスタは次のように変化します。

```
DI  ←DI+1
```

REP (REPZ) またはREPNZと組み合わせた連続サーチの場合は、次のようになります。

```
DI  ←DI+1
CX  ←CX-1
```

これは、ALレジスタとES:[DI]が同じであろうとなかろうと、その結果に関係なく実行されるものです。したがって、SCASB命令でAL=ES:[DI]となった時点（命令実行後）のDIレジスタの値は、+1されていることになります。もし、AL=ES:[DI]となっているDIレジスタを求めたいのであれば、SCASB命令の後で「DEC　DI」を実行しなければなりません。

次に、フラグ変化についてですが、これらの4つの命令はレジスタの増減を伴ったCMP命令をレジスタ単位で行うか、ブロック単位で行うかであり、基本的動作はCMP命令と同じものです。

また、REP(REPZ)またはREPNZと組み合わせた場合には、CXレジスタをカウンターとして、CX←CX−1を実行し、CX=0になるまで連続比較されます。REP(REPZ)ではゼロフラグが立っている間命令が継続され、ゼロフラグがリセットされると終了します。REPNZでは、そのフラグによる動作が逆になります。

試しに、REPNZ(ゼロフラグが立っていなければ命令を続ける）と組み合わせたSCASB命令を考えてみましょう。

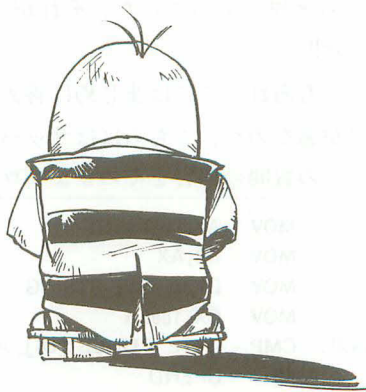
この例では、実行後にゼロフラグが立っていればAL=ES:[DI]となって命令を終了したことになり、ゼロフラグが立っていなければ CX=0となって命令

```
CLD                          ←ディレクションフラグ=0とする
MOV     DI,OFFSET STRING     ←DIレジスタに初期値をセット
MOV     AX,CS
MOV     ES,AX                ←ES=CS
MOV     CX,10                ←サーチ回数をセット
MOV     AL,'A'               ←サーチ文字をセット
REPNZ   SCASB                ←ALの文字をサーチする
```

を終了したと判断できますから、文字'A'の存在の有無をゼロフラグで調べることができるわけです。

なお、CX=CX−1、DI=DI+1は文字'A'の存在にかかわらず、命令の1サイクルで実行されることに注意してください。

次にCMPS命令ですが、これは、SCAS命令がALまたはAXレジスタとES:[DI]とをバイトまたはワード単位で比較したのに対して、AL、AXの部分をDS:[SI]に置き換えて考えることができます。この場合、SIレジスタは自動的に更新されます。

それにしても、いつの間にコンピュータの中に「正マ界」なんていう世界ができたのでしょうか……。

58 SCASB命令の効果的用法

ヤイ、ゲゲゲのQ太郎!! 勝手にウイルス妖怪退治なんて始めるのはやめてくれ。オイラのように正しいコンピュータの発展を願っている者には、はなはだ迷惑な話だぜ。だいたい、おまえはいつも正義面して厚かましいゾ!!
本当の正義の味方とは、オイラのことだ。ウイルス妖怪だって立派なプログラムなんだからな。コンピュータから見れば、有難いお客様だっていうことを忘れないでくれ。あんまりウイルス妖怪をいじめるなら、オイラがウイルスになって戦ってもいいんだぜ。
ところで、ゲゲゲのQ太郎の質問に答えたんだから、オイラの質問にもまじめに答えてくれよな。オイラのはQ太郎と違って、高級な質問だぜ。やはり、ある値があるかどうかを調べるんだが、チェックする値が複数なんだ。
例えば、DATA1:STRINGから、1000Hバイトの範囲で、アスキーコードの'A' 'D' 'F' 'K'のどれかが含まれているかどうか、それを調べようというんだ。
これには、SCASやCMPS命令は使えないよな。やっぱり、CMP命令で1つひとつチェックするしか方法はないだろうな……。
でも、チェックする内容が多くなると、CMP命令というのは結構大変なんだぜ。オイラにもうまい方法を教えてくれ。

ドブねずみ男（邪マ界）

答

ゲゲゲのQ太郎氏から質問があった時、きっとこの手の人物（妖怪）からも質問が来ると思っていました。それが、まさかこんなに早くやって来るとは……。ア然!!

なにはともあれ、ここはまじめに答えなければなりますまい。もはや、誰が正義で誰が悪なのか、こちらにはサッパリわかりませんから。

まず、この質問の内容をそのままプログラム化してみましょう。

```
        MOV     AX,DATSEG1
        MOV     ES,AX
        MOV     DI,OFFSET STRING
        MOV     CX,1000H
CLOOP:  CMP     BYTE PTR ES:[DI],'A'
        JE      CPEND
```

```
        CMP     BYTE PTR ES:[DI],'D'
        JE      CPEND
        CMP     BYTE PTR ES:[DI],'F'
        JE      CPEND
        CMP     BYTE PTR ES:[DI],'K'
        JE      CPEND
        INC     DIEND
        LOOP    CLOOP
CPEND:          ⋮
```

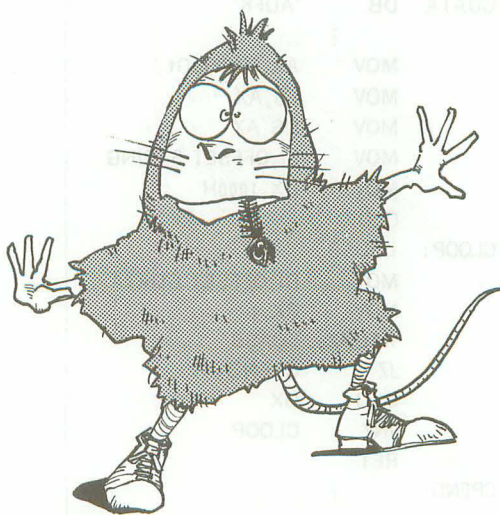
内容のわりには、プログラムが大がかりです。そこで、なぜSCAS命令が使えないという結論を出したのか考えてみましょう。これは、おそらく比較しようとする値('ADFK')がALレジスタになければSCAS命令は使えないと判断したためだと思います。

しかし、ここはコロンブスの卵で発想を転換したいところです。つまり、ALレジスタには DSTA1:STRINGのメモリ内容を入れ、比較しようとする値('ADFK')をメモリ側に置くのです。文章ではわかりにくいので、実際にプログラムを組んでみます。

```
CDATA   DB      'ADFK'
                ⋮
        MOV     AX,DATSEG1
        MOV     ES,AX
        MOV     DS,AX
        MOV     SI,OFFSET STRING
        MOV     DX,1000H
        CLD
CLOOP:  LODSB
        MOV     DI,OFFSET CDATA
        MOV     CX,4
        REPNZ   SCASB
        JZ      CPEND
        DEC     DX
        JNZ     CLOOP
        RET
CPEND:          ⋮
```

こうすれば、たとえチェックする値や順序を変更する場合でも、「CDATA」の内容や並びを変えるだけで簡単に変更できます。チェックする数が多くなればなるほど、このプログラムの簡略さが活きてくるでしょう。

ドブねずみ男氏が、どのような目的でこの複数サーチ・プログラムを利用するのかわかりませんが、あまりコンピュータ内部でモメないようにお願いします。めったに出てこないウイルスより、身近な暴走のほうが困りますから……。

59 SCASB命令とテーブル

コラ〜ッ!! Q太郎にドブねずみ男。ケンカをするでにゃい。少しは、わしの立場も考えてくれにゃ、人間さまがコンピュータを嫌いになってしまうじゃろが。
わしか、わしはこの世界の長老'目ン玉おやじ'に決っておるじゃろ。初代コンピュータが巨大な箱としてこの世に生まれてから数十年、絶えることなくバグを繁殖させ、そしてそれを食べ続け、バグとともに生きてきたのじゃよ。
おまえらみたいに、ポッと出の妖怪とはレベルが違うからのう。ましてや、ウイルスなんぞはわしの歴史に比べたら、ひよっこもいいところじゃ。フォッフォッフォ……。
んにゃから、わしの質問はドブねずみ男なんか問題にならんほど高級じゃ。マ、少し似ておるといえないこともないがの。実は、コールすると押されたキーのアスキーコードを返す「KSCAN」というルーチンがあるんじゃ。その中にな、'A' 'D' 'F' 'K'のどれかが含まれていれば、それぞれに応じたルーチンへジャンプさせようというのにゃよ。

```
CLOOP:   CALL   KSCAN
         CMP    AL,'A'
         JE     CCDTA
         CMP    AL,'D'
         JE     CCDTD
         CMP    AL,'F'
         JE     CCDTF
         CMP    AL,'K'
         JE     CCDTK
         JMP    CLOOP
              ⋮
```

これでは、さすがにSCAS命令は使えないじゃろ。こればかりは、CMP命令で1つひとつジャンプさせるしか手はあるまい。じゃがな、チェックするキーの種類が多くなってくると、これも大変なことなんじゃ……。

目ン玉おやじ（本マ界）

答

とうとう出てきましたか。ウワサには聞いたことがありますが、まさか本物の'目ン玉おやじ'まで登場してくるとは!!

コンピュータの歴史はバグの歴史とも言われていますが、バグとは人間でいうなら痛みのようなもの。もしも、人間が痛みを感じなければ、ケガをしようが骨折しようが、死んでも気がつかないという悲惨な状況になってしまいます。

こんな惨劇を未然に防ぐバグ……。ありがたいバグを育ててくれる'目ン玉おやじ'に心から感謝をしましょう。

さて、今回のようなケースは意外と多く出てきます。問27にあったようなテーブルが利用できればいいのですが、チェックする内容が連続していないので、そのままでは利用できません。

チェックする数が少なければCMP命令でもいいのですが、多くなることも考えて次のような手法があることも知っておくべきでしょう。

```
CDATA     DB      'ADFK'
CTABL     DW      CCDTA,CCDTD,CCDTF,CCDTK

                 ⋮
          STD
CLOOP:    CALL    KSCAN                    ←ALレジスタにキー情報を取り込む
          MOV     DI,OFFSET CTABL+3
          MOV     CX,4                     ←CX=チェックする総数
          REPNZ   SCASB
          JNZ     CLOOP
CCJMP:    SHL     CX,1
          MOV     DI,OFFSET CTABL
          ADD     DI,CX
          JMP     [DI]
```

(注) DS=ES=CSと仮定する

一見すると、CMP命令でジャンプさせていた時より複雑になっているようですが、チェックするキーの内容やジャンプ先が一目でわかるので、デバッグ時などプログラムの内容を簡単に把握することができます。また、同じようなルーチンが多い場合は、「CCJMP」以下を共通ルーチンとして使うことができ、最終的には使用メモリ数も少なくなってくるでしょう。

このようなテクニックはゲームではあまり使うことはありませんが、各種のツールや実用ソフトにおいては結構有益なテクニックです。SCAS命令の活用

法として覚えておくと便利です。

プログラムにバグはつきものですから、結局はそれを速やかにリカバーできるようなプログラムがいいプログラムといえるわけです。

60 ベスト5のチェック

村おこし……といっても、雷おこしの親戚じゃありません。過疎化に悩む地域を活性化し、新しいエネルギーを発生させようという、すばらしい企画のことです。
今でこそ全国どこにでも見られる現象ですが、元祖はここ大分県の一村一品運動なのです。どんなことでも、最初に考えて実行するというのは大変なこと。マシン語のテクニックも同じですね。
そうとわかっていても、マシン語でオリジナルのテクニックを考えるなんて、まだまだ先の話です。とりあえず、現在のテクニックを覚えなければ……。ゲームセンターで、よくスコアのベスト5を表示しているのがありますが、あれはどうやるんでしょう。
例えば、次のようにベスト5があるとします。そして、プレイした人の点数がBXレジスタにある場合、その人の成績が何番目であるか、ベスト5に入るならばベスト5の中に点数を組み入れるようにしたいのです。

```
SCOR1   DW    10000
SCOR2   DW     8000
SCOR3   DW     6000
SCOR4   DW     4000
SCOR5   DW     2000
```

わが村をマシン語村とするため、どうぞよろしく。

村長連合（大分）

答

他人のプログラムを評価するのが趣味のような人がいますが、そういう人に限ってオリジナルのテクニックは考えないものです。どんなにつまらないことでも、最初に考えるのはとても大変です。ましてや、それがブームになるということは、アイデア以上にその先見の明に感心してしまいます。

ゲームセンターで自分のスコアがベスト5に入るのは、名誉の表彰であり、また次にプレイするための刺激剤でもあるわけですが、これも最初に考えた人がどこかにいるはずです。ブームを越えて当然のように処理されているテクニック、これこそが究極のアイデアかもしれません。

この方法は色々あるでしょうが、結局は現在のベスト5と1つひとつ比較していくことになります。次のプログラムは、BXレジスタに現在のスコアを入れてコールすると、ベスト5に入っていればそれを登録し、AXレジスタに順位（6位以下はすべて6）を入れて返すというものです。なお、ベスト5に同じスコアがあった場合は、新しいほうのスコアを上位に入れます。

```
SCORE   PROC
        MOV     SI,OFFSET SCOR1   ←SCOR1から2バイト単位でベスト5が
                                    登録されている
        MOV     AX,1              ←初期順位
        MOV     CX,5              ←登録スコア数
SCORL:  CALL    CPSCO
        LOOPNZ  SCORL
        RET
SCORE   ENDP

CPSCO   PROC
        MOV     DX,[SI]
        SUB     BX,DX
        JB      CPSC1
        ADD     BX,DX
        PUSH    BX
CPSCL:  MOV     [SI],BX           ←新しいベストXを登録し、以前のベスト
        XCHG    DX,BX               X以下のデータを繰り下げる
        ADD     SI,2
        MOV     DX,[SI]
        LOOP    CPSCL
        POP     BX
        XOR     DX,DX             ←ゼロフラグのセット
        JMP     CPSRT
CPSC1:  ADD     BX,DX
        ADD     SI,2
        INC     AX
CPSRT:  RET
CPSCO   ENDP
```

（注）DS＝CSと仮定する

では、ぜひマシン語で村おこしをお願いします。

問 61 1バイト数値ソート

まろは徳川家の流れを祖母の父方の従兄弟（いとこ）の嫁の母の母方の父の祖父の兄の嫁の父の母方の叔父の叔母にもつという、由緒ある身分の生まれ……。
まろは高貴な育ちゆえ、幼少のころはバアヤが毒味をしたあとでなければ食事を許されていなかったのであるぞよ。おかげで、いまだに熱いものはダメ。まれにみる猫舌にてあるぞ。ところが、なんとラーメンが好きで好きでたまらないのよ。
さますとメンはノビる。さまさなければ熱い。このジレンマをなんとか解消したいのであるが、なにか妙手の心得はないかな……。
それはさておき、まろの職業は寺小屋（早い話が塾）の講師にあるぞ。まろは、そこにおいて家庭用電動頭脳（早い話がパソコン）を教えておるが、本当はよくわからんのであるよ。
先日も、子供たちに「マシン語でこの前のテスト結果を成績順に並べてヨ」と言われてしもうたが、難しいのでその日の講義はおひらきにした。100点満点のテストを55人に実施した場合、バラバラになっている点数を成績順に並べるには、どんな方法でやるといいのかのう。
なお、子供たちの点数はメモリの「TENSU」からに入っており、1人1バイトを使用しておる。まろの悩みを聞いてくれることを期待しておるぞよ。

徳ノ川秀麿（島根）

答

まず、猫舌のまろが苦心の末考えた秘伝のラーメンをお教えしましょう。その名は‘氷ラーメン’……。できたてのラーメンに冷蔵庫の氷を4～5個入れるだけでOKです。真夏でも汗をかかずに、熱いラーメンを涼しく短時間で食べられるので、食後の満足感に壮快感も加わって、もう病みつきになること間違いなしです。ぜひお試しください。

満腹になったところで、数値ソートへと移りましょう。数値ソートとは、バラバラになっている数値を大きい順あるいは小さい順に並べ換えるものです。色々な方法が考えられますが、ここに紹介するのは小さい数値から順にフィックスしていくものです。

プログラムの内容は、隣の数値と比較して小さければ入れ換えるという作業をDX回繰り返すことにより、その中の最小値をまず求めます。次に、残された中から最小値を同じようにして求めます。これをDX回繰り返すことで、小

```
SORTS   PROC
        MOV     DX,55-1                 ←DX＝人数－１
        CLD
SOTL0:  MOV     SI,OFFSET TENSU+1
        MOV     DI,OFFSET TENSU
        MOV     CX,DX
SOTL1:  CMPSB                           ←隣の数値と比較し、小さければ入れ
        JNB     SOTS1                     換える
        MOV     AL,[DI]
        XCHG    [DI-1],AL
        MOV     [DI],AL
SOTS1:  LOOP    SOTL1
        DEC     DX
        JNE     SOTL0
        RET
SORTS   ENDP

TENSU   DB      xx,xx,……              ←55人分の点数が連続
```

（注）DS＝ES＝CSと仮定する

さい数値からメモリに固定されてソートが完了するわけです。

１バイトのソートは、色々なソートの基礎となるものですから、プログラムそのものよりもアルゴリズムを把握することが大切です。よくわからない場合は、紙切れに１～９までの数字を書き、バラバラにしてからソートするルールを考えながら並べ換えてみるといいでしょう。

最後に、全国のラーメン屋さんへ、全日本猫舌の会よりお願いです。メニューに'氷ラーメン'を入れてください。

62 2バイト数値ソート

初めてお便りを差し上げますが、私は田舎の中学校で教師をしている者です。昔は田舎の中学校といえば、オンボロ校舎に少ない生徒数というのが相場でしたが、最近は東京の地価高騰のあおりを受けて、都会風のハイカラな校舎に多数の生徒が通っています。
ただ、私のほうがオンボロ校舎タイプの教師なので、本当はこんな堅苦しい言葉が大の苦手です。でも、「……だっぺ」とか「……べェ」と言おうものなら、テレビの影響で標準語化した生徒に笑われてしまいます。実に、ツライ職業……。
そこで、せめて中身だけでも時代の先端を行くようにと、成績管理にパソコンを導入しました。プログラムにも、少しずつですがマシン語を取り入れています。とりあえずの処理として、テストの合計点数を成績順に並べたいのですが、2バイトのソートはレジスタをどのようにヤリクリしていいかわかりません。
次のように「TENSU」から100人分の点数が2バイトずつ並んでいる場合、どのようにしてプログラム化したらいいでしょうか。

```
TENSU    DW   358     ← 100人分の点数が連続している
         DW   125
         DW   532
          ⋮
```

アー、標準語がツライ。どうやったらイイんだっぺ。

田舎教師（茨城）

答

オッ、懐かしいベーベー言葉。やはり、栃城（栃木と茨城）地方は、これが出てこなくちゃダメだっぺナ。伝統の言葉を滅ぼそうとしているのは、やっぱりテレビの影響と東京の地価高騰だべ……。でも、例の語尾上がりのアクセントはまだ健在だっぺ。

それにしても、マシン語がいまだに機種によってバラバラなのは、なんと時代に逆行した現象なのでしょう。もしかすると、コンピュータの開発者というのはテレビを見ているヒマなどないのかもしれません。

とりあえず、本書は8086地方の方言ですべてを表現しなければなりません。

さっそく質問にある2バイトのソートを行うことにしましょう。

```
SORTS   PROC
        MOV     DX,100-1              ←DX=トータル人数－1
        CLD
SOTL0:  MOV     SI,OFFSET TENSU+2
        MOV     DI,OFFSET TENSU
        MOV     CX,DX
SOTL1:  CMPSW
        JNB     SOTS1
        MOV     AX,[DI]
        XCHG    AX,[DI-2]
        MOV     [DI],AX
SOTS1:  LOOP    SOTL1
        DEC     DX
        JNZ     SOTL0
        RET
SORTS   ENDP
```

(注) DS=ES=CSと仮定する

基本的なアルゴリズムは1バイトのソート（問61）と同じなので、今回はプログラムがどのように変化したかを比べてください。

63 ブロック単位の文字サーチ

ビンボー、ビンボー、涙のビンボー。
この次は、どこの家にしようかな……。住みついて楽しいのは、モノスゴイお金持ちが坂道をコロげ落ちるように貧乏になった時だけど、いくら働いても貧乏という家も住み心地はいいもんだぜ。
だけどよ、お金っていうのは決してなくならないからな。ある所からお金が出れば、どこかにそのお金は入っていく。全員貧乏という理想郷は、実現できそうでなかなか難しいもんだわさ。そう考えると、あっしらの存在価値なんて、本当はあるのかないのかわからないんだよナァ。
それに最近は中流意識の強い家が多いから、どこへ住みついても大した感激はないもんだぜ。つまんないから、これからはコンピュータで適当な名前を作って、その名前があればそこに住みつくようにしようと思うんだ。
あっしらの専用ファイルに、8文字単位でインプットされた名字ファイルというのがあるんだ。たとえば、貧田さんという名字の家がある場合、そのファイルにはアスキーコードで「ﾋﾝﾀﾞ____」(_ は スペースを意味する)と入っているんだ。
このファイルをDATA1:NAME 1へロードして(1000_Hバイト分)、適当に作った名前があるかどうかをサーチしようというわけだ。内容は、そのサーチ・ルーチンをコールしたら、DIレジスタで名字の先頭アドレスを、そしてキャリーフラグで名字があったかどうかを返すようなもんがいいな。
おっと、そこから先のあっしの行動はまだ秘密だ。これは、いわゆる企業秘密ってやつだ。親切に教えないと次はおまえの家に住むからな。

貧乏神(貧民峡谷)

答

あれ……?? わが家にはすでに貧乏神が住みついていますよ。ワケもなく新しい貧乏神が来ると、古い貧乏神とケンカになりますぞ。なにしろ、うちの貧乏神様は先祖伝来の大物ですから……。

さて、これは文字列サーチの基本形となるプログラムですから、目的は別にしてグッドな質問といえるでしょう。

プログラムの使い方は、「SDATA」の部分にサーチしたい名字を入れ「SERCH」をコールすればOKです。

```
SDATA   DB    'ヒンダ'            ←名字（8文字）

SERCH   PROC
        MOV   DI,OFFSET NAME1   ←サーチ・オフセット・アドレス
        MOV   AX,DATA1
        MOV   ES,AX             ←サーチ・セグメント・アドレス
        MOV   AX,CS
        MOV   DS,AX
        CLD
        MOV   DX,200H           ←サーチ名字総数（1000H/8）
SCHLP:  MOV   CX,8
        MOV   SI,OFFSET SDATA
CHK8L:  REPZ  CMPSB
        JZ    SCHOK
        ADD   DI,CX             ←次のサーチアドレスにする
        DEC   DX
        JNE   SCHLP
        JMP   SCHRT
SCHOK:  SUB   DI,8              ←名字のある先頭アドレスにする
        STC                     ←名字があったことを示すフラグ
SCHRT:  RET
SERCH   ENDP
```

プログラムを実行すると、サーチしたい名字があればキャリーフラグを立て、DIレジスタをその名字の先頭アドレスにしてもどります。

その後の処理は……、もうおまかせするしかありません。でも、うちに来ても餓死するだけですよ、きっと。

64 連続データからの文字列サーチ

なんみょうなみあむなむあみばてれんめんまちゃあしゅうねぎとろこんぶなむなむかみさまほとけさまあーめんそーめんほうれんそうきつねにたぬきにてんたまおとしがまちゃんちゃがまがまがいものしんしゅんさんそんしゃんそんしょう……。

非常に読みにくいものを紹介しました。これは私が栄えある初代教祖となっている'ましむご教'の経典の頭の部分です。お経はこのような感じで9801文字からできており、信者は毎朝これを祈らなければなりません。

ところが、信者のみならずこの教祖である拙僧も、このお経をまだ全部は暗記できていないのです。なにしろ、深い意味がなさそうでありそうで、やっぱりないのがこの経典の特徴ですから、覚えるのは簡単ではありません。しかし、結果的にはこれも信仰心を高めることに役立っているのです。

というのは、'ましむご教'にはできるだけ速くお教を唱えた者が高い位につくという厳しい戒律があるからです。もちろん、教祖とて油断はできません。そのため、拙僧はこの大経典をコンピュータに入力し、毎日少しでも先まで覚えようとしています。そこで必要なのが、文字列のサーチです。

経典という長い文字列の中から、特定の文字列をサーチするにはどうするべきか。それができないようでは、教祖としての威厳もあったものではありません。なお、サーチする文字列の長さは不定ですが、最大で10文字程度です。

では、なむなむなむ……。

ましむご大僧正（魔心寺）

答

一度でいいから、なってみたいのが教祖さま。どんなにインチキくさい教祖でも、そこに信者がいるのが不思議です。どれほどのご加護があるのかわかりませんが、ましむご教も信者がいるから成立しているのでしょう。

それにしても、このお経と戒律。なんとまァ、ありがたさを感じさせない世紀末的ないい加減さ……。思わず入信したくなります……なんていう人がいるんでしょうか。

とにかく、ここは入信したつもりで希望通りのプログラムを組むことにしましょう。実行方法は「SDVAL」に文字列の文字数（≧１）を入れ、「SDATA」にサーチしたい文字列を入れて「SERCH」をコールします。

```
SDVAL     DB      6                        ←文字数
SDATA     DB      'ナムナムダ'               ←文字列

SERCH     PROC
          MOV     DI,OFFSET OKYOU          ←お経のある先頭アドレス
          MOV     AX,DATA1
          MOV     ES,AX
          MOV     AX,CS
          MOV     DS,AX
          XOR     AX,AX
          CLD
          MOV     CX,9801                  ←お経の総文字数
SCHLP:    MOV     SI,OFFSET SDATA
          MOV     AL,[SI]
          REPNZ   SCASB
          CLC
          JNZ     SCHRT
          MOV     AL,SDVAL
          DEC     AL
          JE      SCHOK
          CMP     AX,CX                    ←CX≧文字列数－1の確認
          JNB     SCHRT
          PUSH    DI
          PUSH    CX
          MOV     CX,AX
          INC     SI
          REPZ    CMPSB
          POP     CX
          POP     DI
          JNZ     SCHLP
SCHOK:    DEC     DI
          STC
SCHRT:    RET
SERCH     ENDP
```

実行後、サーチしたい文字列が見つかれば、キャリーフラグを立ててもどります。DIレジスタは、その文字列のあった先頭アドレスを示しています。このプログラムではサーチできる最大文字列数は10ですが、「SDATA」のワークエリアを多くすればいくらでも長い文字列のサーチが可能です。

なお、プログラムの細かな注意点として、サーチしたい最初の１文字目が見つかった場合、サーチ文字列数が１文字でなければ、お経の残り文字数がサーチする文字数分だけあるかどうかをまず確認しなければなりません。また、２文字目以降で一致しなかった場合には、次のサーチはその場所からでなく、本プログラムのように２文字目を改めて１文字目としてサーチし直さなければなりません。さもないと、サーチ文字列が重なるように含まれている場合（今回の例では「ナムナムナムダ」のような場合）、サーチ洩れを起こすことになります。

これで、教祖の威厳は保てるでしょう。なむなむなむ……。

65 ブロック単位の文字列ソート

発明王トーマス・エジソン……この名を知らぬ者はおるまい。近代科学はすべてこのエジソンの発明があったからこそ生まれたのじゃ。エジソンがいなければ、いまだにコンピュータなどなかったであろう。
わしはエジソンを神と崇拝し、エジソンに次ぐ発明を目指している町の発明家だ。本名を江路苫州（えじ・とます）という。もちろん、両親が付けてくれた本名だ。
町の者は、わしのことをエジソン、いや「えじさん」と呼んでおる。この響きが、わしにとってはたまらない魅力なのじゃ。
さてさて、わしだってエジサンの名に恥じないだけの膨大な発明がある。これまでに発明した名作数は199にものぼっておる。特許や実用新案になったものは、残念ながらまだない。それが、町の発明家のいいところじゃ。ワハハハ……。
すべての発明には16文字以内の名がついておる。そこで、それらをコンピュータに登録してアイウエオ順に並べたいのだが、マシン語は発明ほどには得意でないのだ。
どうやっていいのか教えてくれたら、わしの発明した「灰皿パイプ」をプレゼントしよう。これは灰皿に長いチューブを付け、灰皿側のタバコをチューブでスパスパ吸うものじゃよ。ワハハハ……。

トマス・エジさん（富山）

答

ワハハハ……と言われても、タバコを吸わない人にとっては少しも役に立たないプレゼントです。せっかくですが、プレゼントは辞退しておきましょう。それより、発明の一覧表でも機会があったら見せてください。

ここに紹介するプログラムは、16文字ステップで文字列をソートするものです。原則的には同じ文字列がないことを前提としたプログラムですが、同じものがあってもソートに支障をきたすことはないでしょう。

発明の名前は、セグメント名DATA1、オフセット名NAMEから16文字ごとに置いてあるものとします。したがって、たとえば「灰皿パイプ」であれば、「ﾊｲｻﾞﾗﾊﾟｲﾌﾟ______」(_ はスペースを意味する)のようにトータルで16文字になるように登録するということです。

ソートのアルゴリズムは、問61や問62の数値ソートと同じです。アスキーコードも結局は１バイトの数値ですから、16桁の十六進数を数値ソートしたもの

```
MSORT   PROC
        MOV     DX,199              ←登録してある名前の総数
        MOV     AX,DATA1            ←データ・エリアのセグメント値
        MOV     DS,AX
        MOV     ES,AX
        CLD
        JMP     SOTL3
SOTLP:  MOV     SI,OFFSET NAME      ←名前のある先頭アドレス
        MOV     DI,OFFSET NAME+16   ←2番目の名前のあるアドレス
        MOV     CX,DX
SOTL1:  PUSH    CX
        PUSH    DI
        PUSH    SI
        PUSH    SI
        PUSH    DI
        REPZ    CMPSB
        POP     DI
        POP     SI
        JNB     NOEXS
        MOV     CX,8
EXSOT:  MOV     AX,[DI]
        XCHG    AX,[SI]
        MOV     [DI],AX
        ADD     SI,2
        ADD     DI,2
        LOOP    EXSOT
NOEXS:  POP     DI
        POP     SI
        POP     CX
        ADD     DI,16*2
        LOOP    SOTL1
SOTL3:  DEC     DX
        JNE     SOTLP
        RET
MSORT   ENDP
```

と考えることもできるわけです。

発明というと、つい特許大発明→大金持ちという発想をしてしまいますが、日本のエジさんはどうなんでしょうか。気になるところです……。

66 シフトJISコードからJISコードへ

『今日は東に明日は西……コピー求めて右往左往』
ここでいうコピーとは、宣伝文句という意味のコピーです。複写のほうのコピーではないので勘違いしないでください。冒頭のコピーは、そんなコピーライターの実体を文字で表現したものです。もちろん私のオリジナルコピーです。
カタカナ職業の花形ともいえるコピーライターですが、ハデな外見とは裏腹にハードな仕事です。文学的なセンスや広告の知識が要求されるだけでなく、落雷のようなひらめきが必要なのです。
そんなわけで、私はオンボロのキャンピングカーで'ひらめき'を求めて日本国中をさまよっています。ただいま熊本あたりをさまよい中です。
そこで、気になるのがマシン語による漢字の取り扱いです。今、シフトJISコードをJISコードへ変換したいのですが、どのようにプログラムを組んでいいものか、さっぱりひらめかないのです。なんとかならないものでしょうか。

ピーコー物語（熊本）

答

マシン語も'ひらめき'が勝負です。'ひらめき'とは稲妻……これが落雷するかカラ光に終わるかは、プログラミング技術にかかっています。とはいえ、技術の前にアルゴリズムの壁を崩さないことには光ることもできません。このあたりのバランスは、コピーライターに通じるものがあるのかもしれません。

シフトJISとは、JISコードをシフトして変換し定義されるコードのことです。シフトJISコードからJISコードへ変換するには、次の手順が必要です。

①シフトJISコードの第1バイト≦9F_Hならば、第1バイトから71_Hを引く
②シフトJISコードの第1バイト＞9F_Hならば、第1バイトからB1_Hを引く
③シフトJISコードの第1バイトを2倍して1を加える
④シフトJISコードの第2バイト＞7F_Hならば、第2バイトから1を引く
⑤シフトJISコードの第2バイト≧9E_Hならば、第2バイトから7D_Hを引き、第1バイトに1を加える
⑥シフトJISコードの第2バイト＜9E_Hならば、第2バイトから1F_Hを引く

このシフトJISコードからJISコードへの変換を、具体的にプログラムにすると次のようになります。

```
SHTOJS   PROC
         CMP    AH,0A0H
         JNB    STJ00
         SUB    AH,071H
         JMP    STJ01
STJ00:   SUB    AH,B1H
STJ01:   SHL    AH,1
         INC    AH
         CMP    AL,80H
         JB     STJ02
         DEC    AL
STJ02:   CMP    AL,9EH
         JNB    STJ03
         SUB    AL,1FH
         JMP    STJRT
STJ03:   SUB    AL,7DH
         INC    AH
STJRT:   RET
SHTOJS   ENDP
```

このプロシージャは、シフトJISコードが正常であることが前提ですが、もしエラーコード入力の可能性がある場合には、プロシージャをコールする前に、エラー・チェックをしなければなりません。

おまけとして、JISコードからシフトJISコードへの変換手順も参考のために書き添えておきます。

①JISコードの第1バイトが偶数なら第2バイトに$7D_H$を加算し、奇数ならば第2バイトに$1F_H$を加算する
②第2バイトが$7F_H$以上なら、第2バイトを＋1する
③第1バイトから21_Hを引き、それを2で割る
④第1バイトに81_Hを加える
⑤第1バイトが$9F_H$より大きければ第1バイトに40_Hを加算する

これで、きっと激しいヒラメキが生まれることでしょう。

67 アスキーコードのかけ算（筆算タイプ）

「この石がいいね」と君が言ったから8月6日はハチロク記念日
「32ビットになれよ」だなんて16ビット2個で言ってしまっていいの
キャリーというフラグをキープすることの期限が切れてポップフラグ
ハチロクでいいのというユーザーがいて言ってくれるじゃないのと思う
「AXにムーブしろよ」「ジャンプしろ」といつもいつも命令形でプログラムする君
⋮
わたしはマシン語を自在に使い、流れるようにプログラムする女流歌人です。自己紹介の代わりに、わたしの短歌作品群の中で特に気に入っているものを選んでみました。誰でもできる……と思っていただけるところに、この作品のよさがあります。
どこかで見たような短歌だと思う人もいるでしょう。実は、わたしもそう思っているのです。だから、オリジナルな短歌を目指して、マシン語プログラムのアイデアに磨きをかけています。
例えば、DATAA×DATAB＝DATACというかけ算を、アスキーコードのままで計算できないものでしょうか。例えば、'23'×'45'を次のようにするやり方です。

```
      3233   … '23'のアスキーコード
×     3435   … '45'のアスキーコード
──────────
    313135   … '115'(23×5)のアスキーコード
    3932     … '92'(23×4) のアスキーコード
──────────
  31303335   … '1035'のアスキーコード
```

これができれば、プログラム効率はグンとよくなると思うのですが……。

マシン語歌人・俵まり（岡山）

答

マシン語を口語体のように使いこなし、流れるように美しいプログラムを組むというナゾの女流歌人。そんな歌人のマシン語短歌集『ハチロク記念日』が300万部を超す売行きを示すという時代……になったらいいなと思うハチロク記念日。

マシン語とはアルゴリズムの美を追求する言語です。二進数のかけ算から、アスキーコードのかけ算へとアルゴリズムを変えようとする点に、限られた文

字の世界で美を追求する歌人ならではの美意識を感じます。

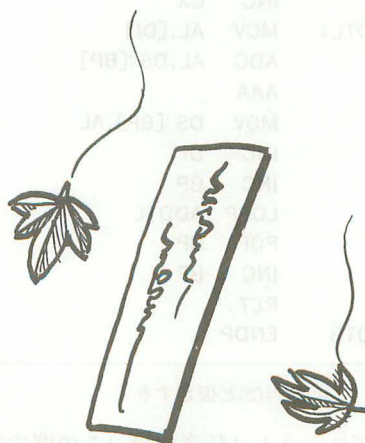
では、このアルゴリズムにしたがってプログラムを組んでみます。

```
KETASU   EQU    2                      ←演算の桁数（任意）
DATAA    DB     33H,32H
DATAB    DB     35H,34H
DATAC    DW     KETASU   DUP(0)        ←計算結果格納用
DATAW    DB     KETASU+1 DUP(0)        ←ワークエリア
DATAN    DW     KETASU

AMULB    PROC
         MOV    DI,OFFSET DATAC
         XOR    AX,AX
         MOV    CX,DATAN
         REP    STOSW
         MOV    BP,OFFSET DATAC        ←加算用ポインター初期値セット
         MOV    BX,OFFSET DATAA
         MOV    CX,DATAN               ←最大桁数（任意）
         JCXZ   AMBRT                  ←最大桁数＝0なら終了とする
AMBL1:   PUSH   CX
         MOV    DI,OFFSET DATAW
         XOR    AX,AX
         MOV    CX,DATAN
         INC    CX
         REP    STOSB                  ←ワークエリアの初期化
         MOV    DI,OFFSET DATAW
         MOV    CX,DATAN
         MOV    SI,OFFSET DATAB
         MOV    DL,[BX]
         AND    DL,0FH
         CALL   MULTB                  ←DATAB×DATAA 1桁
         CALL   ADDTB                  ←演算結果をワークエリアへ加算
         INC    BX
         POP    CX
         LOOP   AMBL1
         MOV    CX,DATAN
         SHL    CX,1
         MOV    AL,30H
         MOV    DI,OFFSET DATAC
AMBL2:   OR     [DI],AL
```

```
        INC     DI
        LOOP    AMBL2
AMBRT:  RET
AMULB   ENDP

MULTB   PROC                    ←DATAAの1桁とDATABとのかけ算を実行
        PUSH    DI
MULTL:  MOV     AL,[SI]
        AND     AL,0FH
        MUL     DL
        AAM
        ADD     AL,[DI]
        AAA
        MOV     [DI],AX
        INC     DI
        INC     SI
        LOOP    MULTL
        POP     DI
        RET
MULTB   ENDP
ADDTB   PROC                    ←かけ算の結果をDATACへ加算
        PUSH    BP              ←加算用ポインター（DATAC）の保存
        CLC
        MOV     CX,DATAN
        INC     CX              ←かけ算の桁数＋1
ADDTL:  MOV     AL,[DI]
        ADC     AL,DS:[BP]
        AAA
        MOV     DS:[BP],AL
        INC     DI
        INC     BP
        LOOP    ADDTL
        POP     BP
        INC     BP              ←加算用ポインターを1桁ずらす
        RET                       （演算結果の10倍に相当する）
ADDTB   ENDP
```

（注）DS＝ES＝CSと仮定する

このプログラムは任意の桁（この例では2桁）を持つアスキーコード間のか

け算の例ですが、基本的なアルゴリズムは、筆算のプロセスをそのまま置き換えたものです。このプログラムで注意しなければばらないのは、DATAAの数値とDATABの数値の桁数が合わない時に、頭に0を付けなければならないということです。まあ、字余りは0で埋めるというわけですから、短歌よりもずっと楽かもしれませんね。

68 4バイトのかけ算（筆算タイプ）

ア　ー、オホン!!
小生はバグというケチな虫でやんす。ご存じのように、小生は姿は見えねど自然発生的に生まれ、適当に暴れ回ったあとは強制的に抹消される運命でやんす。考えると、わびしい運命でやんすネェ。
その代わり、小生には駆除剤とかワクチンのような特効薬は存在しないので、プログラムさえあればいつでも生まれる可能性を持っているんでやんす。神様は、ちゃんと小生みたいな虫けらにも生きる道を与えてくれたんでやんすネ。
ところで、かけ算とは2バイト×2バイトとは限らないすね。4バイト×4バイトのかけ算だって当然必要でやんすから。こんな場合は、どうやったらいいんでやんしょ。
足し算のループでやればいい……なんていう回答なら不要でやんすからね。速度とか美しさなんて小生はどうでもいいんでやんすが、小生の生まれる可能性が少ないプログラムは見てもつまんないんでやんすよ!!

ばぐちゃん（メモリの森）

答

プログラムのないところにバグは発生しませんが、バグもまた生きるためにプログラムを選んでいたとは気付きませんでした。眠たかったり疲れていたりすると、知らず知らずのうちにバグに狙われているのかもしれません。

では、ANSW 1＝DATAA×DATAB をプログラムしてみます。なお、DATAA、DATAB にはあらかじめ4バイトにわたって、数値が格納されているものとします。

```
DATAA   DD    0
DATAB   DD    0
ANSW1   DQ    0
        DW    0          ←計算用ダミーエリア

MULAB   PROC
        MOV   BX,OFFSET ANSW1
        MOV   DI,BX
        MOV   CX,5
```

```
        XOR     AX,AX
        CLD
        REP     STOSW                   ←ANSW1+ダミーの初期化
        MOV     SI,OFFSET DATAA
        MOV     DI,OFFSET DATAB
        CALL    MLBAS
        ADD     DI,2
        ADD     BX,2
        CALL    MLBAS
        MOV     DI,OFFSET DATAB
        ADD     SI,2
        CALL    MLBAS
        ADD     BX,2
        ADD     DI,2
        CALL    MLBAS
        RET
MULAB   ENDP

MLBAS   PROC
        MOV     AX,[SI]
        MUL     WORD PTR [DI]
        ADD     [BX],AX
        ADC     [BX+2],DX
        ADC     WORD PTR [BX+4],0
        RET
MLBAS   ENDP
```

（注） DS＝ES＝CSと仮定する

このプログラムで注意することは、計算結果を格納するエリアANSW1に10バイト確保するということです。また、数値は符号無し（プラスのみ）としています。

これを応用すれば任意の桁数のかけ算が可能になりますが、計算結果の桁数に、2バイトのダミーを加えることを忘れないでください。

69 AX÷CX=BX（小数第一位七捨八入）

水の都ベネチア……。いったい水の都とはどういう意味なんでしょう。水がおいしいのか、それとも川が多いのか、それとも島がたくさん浮かんでいるのか……。不思議な魅力に憧れて、ここベネチアへやって来ました。
まるで海の上に敷かれたような鉄道がベネチアへの入口です。列車の窓から見えるものは、右も左も広大な海原ばかり、いやがおうにも水の都への期待が高まります。やがて列車はサンタルチア駅へ到着です……。
駅前には広場があり歩道もあります。その向こう側には「TAXI」のマークが……。ところが、なんとこれが船なのです。タクシーが船ならバスも船。つまり、車道はすべて船道というわけです。
路地裏には歩道がなくても船道はあります。歩いていけない場所へも、船なら行けるのです。まさに水の都……。
そういえば、日本を出る時にマシン語で AX÷CX=BX（レジスタです）のプログラムをやりかけたままでした。気になるので、やっておいてください。ついでに、小数第1位で四捨五入するようなプログラムもお願いします。なお、数値はすべて正の数です。

旅情の人（ベネチア）

答

ベネチアは潮の香りでいっぱいです。百を超える島々が運河の道路を創り、運河は家と家を結び町を創ります。水の都というより水上都市という感じです。そんなところを旅している旅情の人……。実にうらやましい限りです。

さっそく、気になる AX÷CX=BX をプログラム化してみましょう。一般に、割り算は DIV、IDIV を使いますが、この場合には符号を考えませんから、DIV を使います。当然、CX≠0でなければなりません。

```
WARIS:  CWD
        DIV    CX
        MOV    BX , AX
```

このプログラムでは、先頭でCX=0のチェックをしていませんから、CX≠0が前提となっています。もし、0で割った場合には、除算エラーの割り込みを

生じ、強制的に割り込みルーチンへと処理が移動しますから注意してください。また、演算の結果は、商がAXレジスタに、余りがDXレジスタへと格納されます。

質問にあった、小数第1位で四捨五入(十六進数なので正しくは七捨八入となる)するには、この余り(DXレジスタ)が割る数(CXレジスタ)の半分以上であれば、BXレジスタを+1することで可能となります。注意したいのは、CXレジスタを半分にして余りが出る場合は切り上げるということです(例えば、7なら4とする)。

```
WARIS   PROC
        XOR   DX,DX
        DIV   CX
        MOV   BX,AX
WAEND:  SHR   CX,1     ←CX=CX÷2
        SBB   DX,CX    ←七捨八入を実行
        CMC
        ADC   BX,0
WARET:  RET
WARIS   ENDP
```

「SHR CX,1」による1/2では切り捨て(7なら3)となりますが、その時のキャリーフラグを含めて減算することで、結果的に切り上げた値を減算しています。

これで、安心して旅が続けられることでしょう。

70 アスキーコードの割り算(小数第一位七捨八入)

ニンニンニン……!!
拙者は伊賀の忍者。山を越え、野を越え、時代を越えて、現代社会へやって来たのでござる。得意の忍法は「排気ガス隠れ」……排気ガスに少し細工をして、煙玉で消えたようにするのでござる。
昔の忍者は修行に修行を重ね、必死になって独自の忍法を開発したのでござるが、すでに伊賀には『伊賀忍法トラの巻』があるでござる。それを見れば、こんな忍法なんて簡単なのでござるよ。
そんなことより、今は甲賀忍者のコンピュータに忍び込むほうが大変なのでござる。なにしろ、忍者のコンピュータというのはカラクリだらけ、まるで忍者屋敷のようなワナが待っているのでござる。
下手にプログラムを送ろうものなら、アッという間にやられてしまう。まずは敵の情報を分析しなければならぬでござる。そのためには、アスキーコードで表された数値の割り算ができなければならない。それができないところに拙者の悩みがあるのでござる。
なんとか甲賀に内緒で教えてくれぬでござらぬか……。

影丸(三重)

答

伊賀の影丸……!? 懐かしい名前でござる。拙者は大の影丸ファンでござるゆえ、サインがほしいのでござるが……。

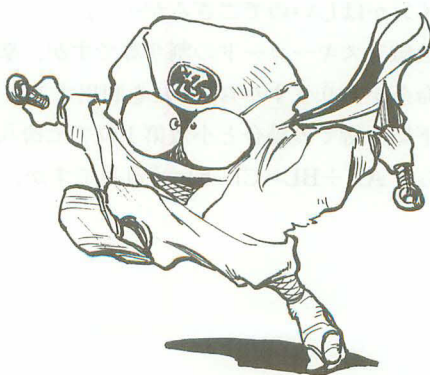
さて、数値アスキーコードの割り算ですが、幸い、8086には割り算用のアスキー補正命令がありますから、これを利用することになります。プログラムは、小数点以下切り捨ての場合と小数第1位で七捨八入した場合とに分けてあります。どちらもAX÷BL=CLという計算ですが、当然BL≠0でなければなりません。

小数点以下切り捨て

```
WARIA    PROC
         AND    AX,0F0FH
         AAD
         AND    BL,0FH
         JE     WRIART
         DIV    BL
         MOV    CL,AL
WRIART:  RET
WARIA    ENDP
```

小数第1位七捨八入

```
WARIA    PROC
         AND    AX,0F0FH
         AAD
         AND    BL,0FH
         JE     WRIART
         DIV    BL
         SHR    BL,1
         SBB    AH,BL
         CMC
         ADC    AL,0
         MOV    CL,AL
WRIART:  RET
WARIA    ENDP
```

問69を読まれた方には、簡単でござったであろう……ニンニン。

71 BX÷CX＝BX.AL（小数第三位七捨八入）

この伊賀者メ……!! 甲賀のコンピュータに無法侵入しようなんて、まだまだ考えが甘い甘い。拙者とておぬしの作るプログラムくらい見当はつくわ。
甲賀にも『甲賀忍法トラの巻』はあるし、伊賀のコンピュータに送り込む刺客プログラムも近いうちに完成するだろう。しかし、伊賀の忍者屋敷もなかなか手ごわいな。先日送った偵察プログラムは、とうとう戻ってこなかった。どうやら、敵のチェックプログラムに破れたようだ……無念。
そこで、拙者は伊賀の計算基準を超すような、正確な割り算プログラムで対抗しようと思うのだ。つまり、16ビットの割り算を小数第2位まで求めようというのだ。レジスタ構成としては、次のようなものを考えている。

BX÷CX＝BX.AL

これまでは、商は整数だったから誤差がどうしても大きくなる。計算結果をCX倍しても元のBXとは相当違った値になる可能性を否定できなかった。これが小数第2位までになると、ほぼ正確に元の値に戻すことができるのだ。

（例）四捨五入による計算
50÷30＝2 → 2×30＝60 ………………誤差が大きい
50÷30＝1.67 → 1.67×30＝50.1…………誤差が小さい

これをなんとか実用化したい。伊賀に内緒で頼む。

サスケ（滋賀）

答

少年忍者サスケ……。拙者はサスケのファンでござる。ここに登場したサスケ氏はかなり大人びていますが、それでもやはりサスケはサスケ。ぜひサインをください。←なんと節操のない人物!!

それにしても、コンピュータをいじる影丸とサスケ……いったい現代の伊賀と甲賀の忍者は何を考えているのでしょうか。割り算がターゲットになる理由もさっぱりわかりません。とにかく、質問通りにプログラムを組むことにします。割る数（CX）のゼロチェックはしていませんから、その恐れがある場合は計算前にそのチェックをしてください。

```
WARIX   PROC
        MOV    AX,BX
        MOV    DX,0
        DIV    CX
        MOV    KEPAX,AX       ←KEPAX に整数部の商を求める
        XOR    AX,AX
        MOV    AH,DL
        MOV    DL,DH
        MOV    DH,AL          ←DX:AX に余り×100H を求める
        DIV    CX
        MOV    BX,KEPAX
        SHR    CX,1
        SUB    DX,CX
        CMC
        ADC    AL,0           ←小数第3位の七捨八入計算
        ADC    BX,0
        RET
WARIX   ENDP

KEPCX   DW     0
```

（注）DS＝CS と仮定する

整数部の商は簡単に求められますが、小数部はレジスタの役割とアルゴリズムを理解しておかないとわかりにくいかもしれません。順を追って確認しましょう。整数部の計算が終わった時点で、レジスタの内容は次のようになっています。

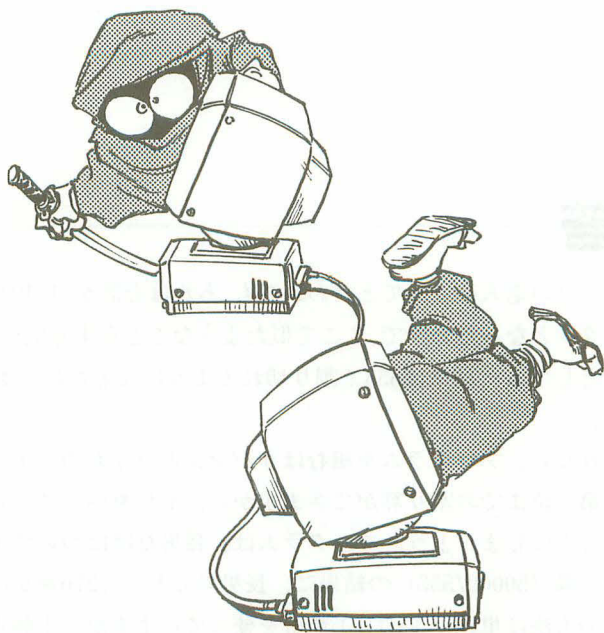
AX レジスタ＝商（整数部）
DX レジスタ＝割り算の余り
CX レジスタ＝割る数

小数点以下の割り算は、余り部分を100_H倍して再び割り算をして求めています。なお、ここで求められる小数部の商は十六進数ですから、分数でいうなら1/256単位の数値となります。

最後に、小数第3位の七捨八入処理をします。七捨八入した結果によっては

ALレジスタが桁上がり（$FF_H \rightarrow 00_H$）するかもしれませんので、その場合には整数部（BXレジスタ）が＋１されるよう配慮しなければなりません。

忍法同様、最後の最後まで気をゆるめないでください……。

72 BX.AL×BP=AX（小数第一位七捨八入）

その昔、アラブの砂漠で３人の男が遺産相続でモメていた。亡くなった父親の遺言によると「遺産のラクダは、長男が1/2、次男が1/4、三男が1/6、仲良く分けろ」とあったそうだ。ところが、ラクダは11頭しかいない……。
今にもなぐり合いが始まろうという時、一人の老人が現れ「お若いの、わしの持っているラクダを一頭やるからケンカはやめるがいい」とラクダをくれた。三人は喜んで、12頭のラクダを遺言通りに分けた。

長男：12×1/2＝６
次男：12×1/4＝３
三男：12×1/6＝２

分けてみると一頭余っている。分配に満足した３人は、そのラクダを丁重に老人に返したそうだ。老人は、ニヤリとしていずこかへ去っていったという。

……こんな話をしながら、父がボクたちに50000円ものお年玉をくれるというのだ。ただし、長男が270/555、次男が180/555、三男が105/555となるように、マシン語を使って分配するようにと命令された。
長男のボクとしては、なんとしてもこのプログラムを完成させなければならない。どうか、アラブの老人になってください。

総領の甚六（高知）

答

アラブの老人になってと言われても、あれは合計が11/12だったからメデタシメデタシとなったわけで、ここで似たようなことをする気分になって505円を出したとしても、結果は555で割り切れるようになるだけでお金は戻ってこない……。

それなら、プログラムを組むほうがマシというもの。すでに、問71によって小数第２位までの割り算ができますから、それをベースに各人の金額を求めてみることにしましょう。プログラムは、長男だけについて示しています。

割り算（50000/555）の結果に、長男の分として270をかけるわけですが、かけ算の方法は単純に（MUL）命令を使っていますが、小数点以下の処理が増え

```
MAN01  PROC
       MOV    BX,50000
       MOV    CX,555
       CALL   WARIX        ← BX÷CX=BX.AL（問71を利用）
       CBW
       MOV    BP,270       ← BP=かける数
       MUL    BP           ← DX:AX=AX×BP
       MOV    CX,100H
       DIV    CX           ← AX=DX:AX÷CX
       CMP    DX,080H      ←最後の七捨八入
       JB     MAN02
       INC    AX
MAN02: MOV    CX,AX
       MOV    AX,BX        ← AX=BX
       MUL    BP           ← DX:AX=AX×BP
       ADD    AX,CX
       RET
MAN01  ENDP
```

た分だけレジスタが不足します。そのため、ここではBPレジスタを活用しています。

最終的な計算結果は、小数第1位を七捨八入したものがAXレジスタに入ります。

《 計算結果 》

長男：24324　（AX：5F04$_H$）

次男：16216　（AX：3F58$_H$）

三男：　9459　（AX：24F3$_H$）

合計では49999円と割り切れなかった分の誤差が出ますが、小数点以下を考慮しない計算に比べればはるかに高精度です。もしも残った1円でモメるようであれば、最後の七捨八入を五捨六入（「CMP　DX,80H」→「CMP　DX,60H」）とすれば、三男の取り分だけが＋1されて合計でピタリ50000円になります。

現代版アラブの老人……。それは、コンピュータという砂漠に生きるマシン語プログラムのことかもしれません。

73 アスキーコードからBXレジスタへ変換

西暦2100年……。
1900年代後半に出回っていた8086は、すでにその役目を完全に果たし、CPUとして成すべき使命を全うしたかに思われていた。それどころか、その存在はコンピュータ史研究家の間でも忘れられた存在となっていた。
なにしろ、子供向けパソコンでさえ1024ビットCPUを8個搭載している時代だ。立体テレビによる完全3D画面が、まるでビデオ映像のようにパソコンで展開されている。しかも、それは家庭でのゲームである。ゲームセンターは亜空間体験ゲームルームとなり、プレイヤーは完全に立体虚像化された空間を自由にさまよったり、勇者として悪を倒すという古典的シナリオの世界で実際に戦ったりできるのであった。
私は、ドラゴラン銀河系ツバイシュタイン星タイムトラベラーである。地球の未来を見てきたので、正直に報告しておこう。
……で、私のタイムマシンであるが、メインCPUはなんとその8086なのである。ところが、どうもプログラムにバグがあるらしいのだ。アスキーコードで入力された年代（画面上の数字）を、マシン語プログラムで操作するために2バイトの数値に変換しなければならないのだが、それがうまくいっていないようだ。
ところが、残念ながら私は8086マシン語はよくわからない。未来の情報を教えた代わりに、8086でのプログラムを教えてもらいたい。

アストロ・ベイダー（TWS星）

答

テレビ画面の中で実際にプレイをする……これは、まさに究極のゲームといえるでしょう。もっとも、それが実現した時には、すでにそれ以上のゲーム欲が人間を支配しているのは間違いないでしょうが……。

平面型テレビの次は立体テレビとなると考えられていますが、いくら未来のテレビでも実像テレビ（テレビの中の料理が食べられる）だけは無理でしょうね。となると、虚像空間の次は何が出てくるのか、そのあたりの情報も知りたいものです。

未来はともかく、当面はアスキーコードで示された数字を2バイトの数値に変換するという現実的なプログラムを完成させなければなりません。これは、

キースキャン結果がアスキーコードで返される場合や、テキスト画面上に表示されている数字を数値に変換するという場合にも必要な基本テクニックです。

```
ADATA   DB     ' ','12345',' '           ←データはスペースで囲まれている

ATOHL   PROC
        MOV    DI,OFFSET ADATA+1         ←DI=データの先頭アドレス
        MOV    AL,' '
        MOV    CX,0FFFFH
        CLD
        REP    SCASB
        DEC    DI                        ←DI=データのエンドサイン（' '）アドレス
        XOR    BX,BX                     ←BX=十六進数に変換後の値
        MOV    CX,1
        CALL   AREAD
        MOV    CX,10
        CALL   AREAD
        MOV    CX,100
        CALL   AREAD
        MOV    CX,1000
        CALL   AREAD
        MOV    CX,10000
        CALL   AREAD
        RET
ATOHL   ENDP

AREAD   PROC
        DEC    DI
        MOV    AL,[DI]
        CMP    AL,' '
        JE     AREND
        SUB    AL,'0'-1                  ←アスキーコードを「数値＋1」に変換
ARDLP:  DEC    AL
        JE     ARRET
        ADD    BX,CX
        JMP    ARDLP
AREND:  POP    AX                        ←SP合わせのダミー
ARRET:  RET
AREAD   ENDP
```

（注）DS=ES=CSと仮定する

このプログラムでは、' '(スペース)をデータの前後に存在させることで数字の区切りとしていますが、これはケースバイケースで自由に変更することができます。また、変換できる最大数字数は5桁です。5桁以上の数字列の場合は、後半の5桁が有効数字となります。

ただし、2バイト(0~65535)を超えるかどうか、あるいはデータに数字以外の文字があるかどうか等、異常事態のチェックはしていません。キースキャンデータの場合であれば、少なくとも数字の上下('0'~'9')くらいはチェックしたほうがいいでしょう。

これでタイムマシンが直ったなら、西暦2200年あたりのコンピュータ事情も調べてきてください。

74 アスキーコードからBCDの数値へ変換

こちらはツバイシュタイン星よりタイムマシンの製造依頼を受けたトライシュタイン星のトヨサン時動社です。実は当社のタイムマシンにバグが発見されたのですが、すでにツバイシュタイン星のタイムトラベラーは貴地球に向かってしまった後でした。
このままでは、時空間をフラフラする危険性があるので、ぜひ入力した年代をマシン語上で使用できるように修正したいのです。
プログラムの内容は、アスキーコードで入力された年代をBCDによる数値データに変換するというものです。もし、間違って2バイトの数値などに変換してしまうようなことがあると、ますます時空間の狂った世界へワープしてしまうでしょう。
ツバイシュタイン星の話では、乗務しているタイムトラベラーはマシン語が弱いとのことなので、おそらく地球にて誰かに質問するはずだということでした。きっとこの情報を受信できるような機関へ質問していると思われるので、そのような質問があったら次のように修正をするように連絡してください。
では、プログラムを送りまままままままーすすすす。ややややや、ははははは発信機ががががが、こここここ故障ししししてててまま………。

技術部長ノホホン（TRS星）

答

運の悪い時はすべてがスレ違いになってしまうもの。タイムマシンのバグ発見が遅れたことで、運に見放されてしまったのかもしれません。ほんの少し前に、ツバイシュタイン星のタイムトラベラーから2バイトの数値に変換する質問があったばかりです。

こうなったら、運が好転することを期待するしかないでしょう。もしかするとまだ今の年代をうろついている可能性もないとはいえません。とにかく、アスキーコードの数字列をBCDに変換するプログラム、これを急いで作ることにします。

```
ADATA   DB      ' ','12345',' '     ←データはスペースで囲む
BCDDT   DB      0,0,0               ←BCDに変換された値が入る

ADBCD   PROC
```

```
        PUSH    DS
        PUSH    ES
        MOV     AX,CS
        MOV     DS,AX
        MOV     ES,AX
        MOV     DI,OFFSET ADATA+1
        MOV     AL,' '
        MOV     CX,0FFFFH
        CLD
        REPNZ   SCASB
        DEC     DI                      ←DI=データのエンドサイン(' ')アドレス
        MOV     BX,OFFSET BCDDT         ←BX=BCDに変換された値が入るアドレス
        MOV     CX,3                    ※
        XOR     AX,AX
        XCHG    DI,BX
        REP     STOSB                   ←BCDデータエリア・クリア
        XCHG    DI,BX
        DEC     BX                      ←※(BX=BCDDT+2となっている)
        MOV     CH,3
ADBL1:  DEC     DI
        MOV     AL,[DI]
        CMP     AL,' '
        JE      ADBRT
        MOV     CL,4
        SHL     AL,CL
        DEC     DI
        MOV     AH,[DI]
        MOV     CL,4
        SHR     AX,CL
        MOV     [BX],AL
        CMP     BYTE PTR [DI],' '
        JE      ADBRT
        DEC     BX
        DEC     CH
        JNE     ADBL1
ADBRT:  POP     ES
        POP     DS
        RET
ADBCD   ENDP
```

BCDに変換された数値は「BCDDT」からに入ります。このプログラムではメモリを3バイトしか用意していませんが、メモリさえ確保すれば、BCDによる数値は桁の制限がありません。桁を増やす場合は、BCDのメモリ数を意味するレジスタの値（※印2箇所）も変更してください。

ここで、アスキーコードからBCDの数値に変換している処理に着目してみましょう。アスキーコードの数字（30_H～39_H）は下位4ビットがBCDに必要な部分ですから、1バイトにつき2つのデータが必要になります。注意しなければならないのは、数字列の総数が奇数の場合です。ここでは、ダミーとしてスペース（20_H）でBCD変換を終了させていますが、スペースの下位4ビット＝0であることが、このダミー処理をうまく成立させているのです。

せっかく作ったプログラムですが、肝心の質問者は不明、連絡先も不明……。事情はよくわかりませんが、バグだけは今も未来も変わらぬ存在のようです。

75 BCDの数値をアスキーコードへ変換

ハイ、ちょっとそこのタイムマシン止まりなさい。ジグザグワープは銀河交通法で禁じられていますヨ……オイ、いつまでフラフラしたワープを繰り返すのだ。もしかして、酔っているのか。酒酔いワープは即１年間の免停と罰金20万ゴールドだぞ!!
アッ、また不法なワープをした。どこへ行くか、コラ……『銀河タイムパトロール隊員の白いタイムマシン』通称白タイの命令には絶対服従という法律を忘れたのか。この法律は、各星が集まってできた銀河連邦の「時空ワープに関する交通法規特別準備委員会」によって正式決定された由緒ある基本六法の１つだぞ!!
オ、やっと止まった。なんだ、年代のアスキーコードをBCDに変換するプログラムを直していたのか。仕方ない、ジグザグワープの件は許そう。だが、ワープ先の年代はちゃんとタイムマシンの前後に表示しなきゃダメじゃないか。
その方法……？　それはBCDの数値をアスキーコードに変換して、指定のメモリに入れてやればいいのさ。ン、マシン語がわからない……!?　そんなムズかしいこと、オレに聞かないでくれよな。
だ、誰か……この会話を傍受していたらプログラムを組んでやってくれないか。そうしないと、オレも見張りとしてここを動けない……トホホ。

銀河のシェリフ（地球出身）

答

どうやらアスキーコードをBCDに変換するというプログラム(問74)は届いたようですが、新たにその逆の問題が起きているようです。

単純に考えれば、入力されたアスキーコードをそのまま使えばいいということになりますが、色々事情があって別のプログラムにしているのでしょう。このあたりはタイムマシンの設計者（本書の著者ではない）の意思を尊重するしかありません。

BCDの数値を上位／下位に分ける方法は問48にもありましたが、プログラム的にもう少し簡単な方法があります。問74にあったプログラムを逆用すればいいのです。

CLレジスタの値はシフト命令の実行後も不変ですから、ループに入る前にCLレジスタの初期値を４としています。また、ALにBCDコードの上位が、

```
ADATA   DB     0,0,0,0,0,0          ←アスキーコードに変換された結果
BCDDT   DB     12H,34H,56H          ←BCDによる数値

BCDAS   PROC
        MOV    SI,OFFSET BCDDT      ←BCDによる数値のあるアドレス
        MOV    DI,OFFSET ADATA      ←アスキーコード数字が入るアドレス
        MOV    DX,3                 ←BCDDTのバイト数
        MOV    CL,4                 ←BCDをアスキーコードに変換
BCALP:  XOR    AX,AX
        MOV    AH,CS:[SI]
        ROL    AX,CL
        SHR    AH,CL
        OR     AX,3030H
        MOV    CS:[DI],AX
        ADD    DI,2
        INC    SI
        DEC    DX
        JNZ    BCALP
        RET
BCDAS   ENDP
```

AHにBCDコードの下位が入るように工夫している点にも注意してください。

これで、白タイの隊員も無事拘束から解放されることでしょう。

76 BCDをシフトする

オー、ワタシハ　ニホゴ　ヘタナ　ガイジ　デース。チョド　イマ　ニホノ　ゲンカン　ナリタヘ　ツイタバカリデース。
ワタシ　ウマレハ　フランス　ソダチハ　ブラジル→アメリカ→インド→ドイツ→サウジアラビア→オランダ→エチオピア→中国→ペルー→モンゴル→ケニア→日本→ポーランド→イタリア→エジプト……。ソノアトハ　キオクニ　アリマシェーン。
ダカラ　マトモニ　シャベレル　コトバ　ナーニモナイ。　セメテ　マシゴ　カンゼンニオボエタイネ。　ソコデ　シツモンスルアル。
レジスタノ　アタイヲ　2バイスルトキ「SHL　AL,1」トカ　シフトシマスネ。　アレッテ　BCD ニモ　ツウヨウシマスカ？
タトエバ、AL＝5ヲ　2バイスル　プログラムハ　コレデ　OK　デスカ。

```
MOV   AL,5
 SHL  AL,1
DAA
```

ドゾ　ヨロシク　オネガイ　モシアゲマス。

ドコデモガイジン（千葉）

答

ドモ　テイネナ　シツモン　アリガトゴザマス。　モシワケナイケド　フツノ　ニホゴデ　カカセテモライマス。

ふぅ〜っ……。カタカナに気を取られて、質問の内容を忘れるところでした。それにしても、さすがに多国を渡り歩いているだけあって質問も鋭い点を突いています。というのは、このプログラムは正解のようで正解でないし、間違っているけども結果は合っているという妙なものなのです。

まず、これを実行した結果はどうなるかというと、ALレジスタ＝10_Hとキチンと2倍された値が得られます。したがって、ここでの値だけを見れば不正解とはいえません。しかし、これをAL＝8_Hで実行したらどうなるでしょうか。

結果がAL＝16_Hとなってくれればいいのですが、実はこれもAL＝10_Hとなっ

てしまうのです。つまり、結果が正しくなる時もあるし狂う時もあるのです。当然、これは間違った使い方ということになります。

DAA命令というのは、あくまでもBCD数値の加減算命令を実行した後にだけ正常な働きをする命令です。しかし、それ以外の場合でも無意識（？）にALレジスタの値をBCD化しようとする実直な命令なのです。その結果、偶然にも期待した値になることもあるし、まったく違った値になることもあるわけです。したがって、このプログラムは次のように改めなければなりません。

```
MOV   AL,5
ADD   AL,AL
DAA
```

ちなみに、左シフトして2倍になるとよく言いますが、これは正確には二進数を左シフトすると二進数表記で10倍（十進数表記で2倍）になるということです。つまり、何進数であれ左シフトすればその表記での10倍になっているのです。試しに我々が日常使用している十進数で考えてみましょう。

1 → 10 → 100 → 1000 → 10000 → 10000
5 → 50 → 500 → 5000 → 50000 → 50000

実に当り前のことですね。BCDというのは二進化十進数、つまり考え方としては十進数です。シフトさせるのであれば4ビット単位でシフトさせなければなりません。当然、結果は十進数で10倍となり、DAAをする必要はありません。参考までに、3バイトのBCD数値の左シフトを行ってみましょう。

```
DTBCD     DB      00,12H,34H          ←BCDによる数値

SFT10     PROC
          MOV     SI,OFFSET DTBCD
          MOV     DX,2                ←BCD数値のバイト数－1
          MOV     CL,4
LOOPS:    MOV     AX,CS:[SI]
          XCHG    AL,AH
          SHL     AX,CL
```

```
        MOV     CS:[SI],AH
        INC     SI
        DEC     DX
        JNZ     LOOPS
        MOV     CS:[SI],AL
        RET
SFT10   ENDP
```

プログラム実行後には、DTBCD＝01_H、23_H、40_Hとなります。蛇足ですが、100倍ならメモリ単位のブロック転送で処理できますし、右シフトをすれば結果は1/10となります。

デハ　ドゾ　マシゴデ　セカイノコクサイジンニ　ナテクダサ〜イ……。

問77 BCD 数値の四捨五入

ウ ワッ!!

また仲間がやられてしまった。今度の相手はとても強い……。だいたい、アイツら4人組だもんナァ。魔法を使うヤツもいるし、ケガを治すヤツもいる。おまけに戦うたびにますます強くなっていくようだ。

これじゃ、いくらこちらが頑張ってもかなうわけないよナ。おや、マタンゴおやじがイイ線いってるぞ。うまいことアイツらを眠らせたようだ。そこだ、行けっ!! なぐれ、パンチだパンチ!! そうだ、ボディ……ボディ!!

やったァ〜。全員倒したぞ〜。これで、アイツらの顔は二度と見ることはない。平和な世界が戻った……トト、と思ったらアイツら生き返っちまった。

なんてイイ加減な世界なんだ。ここはデジタルの世界のはずなのに、まるで四捨五入の世界みたいだ。

そういや、BCD の数値を四捨五入するってのはできるんだろうか。たとえば、メモリを3バイト（6桁）使って、前半の5桁を整数部、残り1桁を小数部とし、小数第1位四捨五入なんていうのはできるんだろうか。

どうせ、オレたちの存在なんてゴキブリ以下なんだろうけど、それくらいはハッキリしておきたいよナ。アイツらいくらでも生き返れるからいいけど、オレたち死んだらしまいのワビしい人生なんだからサ。

スライムちゃん（テレビ界）

答

グチを言いたいのか質問をしたいのか、本音は不明ですが、勇者の一人としてスライムちゃんの気持ちがわからないわけでもありません。かなりハデに暴れ回った経験がある以上、せめてもの罪ほろぼしに、無条件で BCD の四捨五入を教えることにしましょう。

四捨五入とは、4以下切り捨て5以上切り上げですから、＋5をして9以下切り捨てと考えることができます。ですから、小数第1位四捨五入なら、0.5を足して小数点以下を切り捨てればいいのです。

《 小数第1位四捨五入の例 》

12.3 → 12.3＋0.5＝12.8 → 12.0

12.7 → 12.7＋0.5＝13.2 → 13.0

では、質問にあるように3バイト（6桁）のBCD数値の1桁目で四捨五入するプログラムを組んでみます。

```
DTBCD   DB      12 H,34 H,56 H          ←BCDデータ

SISYA   PROC
        PUSH    DS
        MOV     AX,CS
        MOV     DS,AX
        MOV     BX,OFFSET DTBCD+2
        MOV     AL,[BX]
        ADD     AL,5                    ←四捨五入のため＋5する
        DAA
        PUSHF                           ←キャリーフラグを退避
        AND     AL,11110000 B           ←1桁目をゼロにする
        MOV     [BX],AL
        DEC     BX
        POPF
        MOV     AL,[BX]                 ←以下、通常のBCD計算
        ADC     AL,0
        DAA
        MOV     [BX],AL
        DEC     BX
        MOV     AL,[BX]
        ADC     AL,0
        DAA
        MOV     [BX],AL
        POP     DS
        RET
SISYA   ENDP
```

四捨五入するのは1桁目ですが、桁上がりがあるかもしれませんから、計算は全桁にわたって行わなければなりません。

ちなみに、単なる切り上げは零捨一入のことですから、＋9をして切り捨てをすればいいわけです。もちろん、二捨三入でも八捨九入でも自由自在です。いかにも、スライムちゃんにふさわしいプログラムではないですか……。

78 BCD数値×AL

先日、友人と3人でひなびた民宿へ泊まった。1人一泊1000円という安さだった。次の日、ぼくは友人2人から1000円ずつ集めて、まとめてお金を払おうとした。すると、バァさんが「また来ておくれ」と言って500円サービスしてくれた。3人で500円じゃ割り切れないので、ぼくはこっそり200円ネコババした。そして、残りをみんなで100円ずつ分けた。結局、1人900円で泊まったわけだから、3人で2700円だ。それに、ぼくのネコババした200円を足すと……。アレッ、2900円しかない。確か、最初は3000円あったはずだ。どこかに落としたのだろうか。

ぼくは、コンピュータでこの謎を解決しようと思う。数値はBCDで3バイトを使うことにした。そうだ、色々な人数でも通用するようにしたほうがいい。ぼくは、人数はそれほど多くならないからALレジスタで示すことにした。だから、ALレジスタは十六進数の数値ということになる。

ということは、『3バイトのBCDの数値×AL』ができればいいのだな。ぼくは、いつものように独り言をつぶやきながら、プログラムを組むことにした。でも、BCDのかけ算ができなかった……。

セブンっ子（鳥取）

答

数のマジックという言葉がありますが、とかく人間は数にダマされやすいものです。朝三暮四とは猿のこと、なんて思っていると痛い目に会うかもしれません。

8畳の広い和室←畳そのものが極端に小さい
3個で9割引←3割引×3個
定価10000円を980円←500円くらいの商品に10000円の定価を付ける
合格率100%の予備校←1人が5校に合格すると4人が不合格でも合格率は100%

時ソバ、ねずみ講、減税と増税……、わかっていてもついついダマされそうです。もっとも、今回のネコババの計算は自業自得ですが……。

ネコババ計算の解決は別として、『3バイトのBCDの数値×AL』という計算は正しくプログラムしなければなりません。ここでは、ALレジスタの値がBCDではなく十六進数として扱われているので、足し算のループでかけ算を実行します。

```
KEKKA   DB      0,0,0               ←かけ算の結果が入る
DTBCD   DB      01H,23H,45H         ←BCDの数値

KAKBA   PROC
        MOV     DI,OFFSET KEKKA     ←結果エリアのクリア
        MOV     CX,3
        MOV     DL,AL               ←DL＝ループ回数（かける数）
        XOR     AX,AX
        REP     STOSB
        AND     DL,DL
        JE      KAKRT
        MOV     DI,OFFSET DTBCD
        MOV     BL,[DI]             ←BL＝(DTBCD)
        MOV     BH,[DI+1]           ←BH＝(DTBCD＋1)
        MOV     AH,[DI+2]           ←AH＝(DTBCD＋2)
        MOV     DI,OFFSET KEKKA     ←加算ループによるかけ算
        MOV     CL,DL
        MOV     CH,0
KBAL1:  MOV     AL,[DI+2]
        ADD     AL,AH
        DAA
        MOV     [DI+2],AL
        MOV     AL,[DI+1]
        ADC     AL,BH
        DAA
        MOV     [DI+1],AL
        MOV     AL,[DI]
        ADC     AL,BL
        DAA
        MOV     [DI],AL
        LOOP    KBAL1
KAKRT:  RET
KAKBA   ENDP
```

（注）DS＝ES＝CSと仮定する

かけ算の結果は「KEKKA」からの3バイトに入ります。BCDによる最も単純な計算例ですから、数のマジックでゴマ化されることもないでしょう。

79 BCDどうしのかけ算（筆算タイプ）

から っ風とカカア天下……とくりゃ、ウチの母ちゃんにピッタリ。暑さ寒さもなんのその、強くたくましく頼りになる母ちゃんだ。
北風がビュウビュウ吹いたって、絶対に倒れないドッシリした体格。ワシが安心して働けるのも、この母ちゃんがいるからだ。給料が安くても文句は言わない。おまけに、料理はうまいし運転もできる。
母ちゃんが病気になってもワシは抱えられないが、ワシが病気になると母ちゃんはヒョイと抱えてくれるんだぜ。どうだい……カカア天下はいいだろう!!
でもな、いくら母ちゃんでもコンピュータだけはダメだ。これだけは教えてもらうことができない。仕方ないので質問だ。
実は、BCDでかけ算をやってみたいのだ。もちろん、問67にあるようなカッコイイ方法でだ。全部を足し算のループでグルグル回すなんていうのはダメだぞ。なにしろワシはソフトハウスの部長という肩書だからな。
おっと、レベルの低いソフトハウスだなんて思わないでもらいたい。ワシは営業部長だからプログラムは関係ないんだ。これはあくまでワシの趣味だ、し・ゅ・み……。
信じられないんだったら、母ちゃんに聞いてみな。

ワシは父ちゃん（群馬）

答

なになに……「追伸　書き忘れたけど、母ちゃんは結構美人なんだぞ……」だって。こうなると、質問というより母ちゃんの宣伝みたいな気もします。

なにはともあれ、すべてを信じてBCDのかけ算をカッコヨクやることにしましょう。ただし、問67の場合のようにプログラムを組むことはできません。なぜなら、MUL命令を使うためにはニブル単位（4ビット単位）のデータをバイト単位に展開しなければならないからです。

問76でも説明したように、BCDでは数値を4ビット単位で扱うことが原則です。したがって、BCDのかけ算は筆算による十進数のかけ算と同じような感覚でプログラムを組むことになります。

《プログラム的筆算によるかけ算の例》

```
    123
×  456
   738 ← 123×6  (足し算のループで行う)
  6150 ← 1230×5 (足し算のループで行う)
  492  ← 123×4  (足し算のループで行う)
 56088
```

では、これを実際にプログラミングしてみましょう。このプログラムでは、ADATA（3バイト：十進数で6桁）× BDATA（3バイト：十進数で6桁）＝KEKKA（6バイト：十進数で12桁）の計算を行っています。

```
KEKKA   DB      0,0,0,0,0,0
DUMMY   DB      0                   ←ADATAを4バイトとして扱うためのダミー
ADATA   DB      0,01H,23H
BDATA   DB      0,04H,56H
ADA10   DB      0,0,0,0             ←ADATA×10が入る

KAKEX   PROC
        PUSH    DS
        PUSH    ES
        MOV     AX,CS
        MOV     DS,AX
        MOV     ES,AX
        MOV     DI,OFFSET KEKKA
        XOR     AX,AX
        MOV     CX,3
        REP     STOSW               ←結果をクリア
        MOV     SI,OFFSET ADATA
        MOV     DI,OFFSET ADA10+1
        MOV     CX,3
        REP     MOVSB
        MOV     DI,OFFSET ADA10
        MOV     DX,3                ←BCD数値のバイト数
        MOV     CL,4
LOOPS:  MOV     AX,[DI]             ←ADATA×10をADA10に用意する
        XCHG    AL,AH
        SHL     AX,CL
```

```
        MOV     [DI],AH
        INC     SI
        INC     DI
        DEC     DX
        JNZ     LOOPS
        MOV     [DI],AL
        MOV     SI,OFFSET BDATA+2     ←SI＝かける数のエンドアドレス
        MOV     BX,OFFSET KEKKA+5     ←(KEKKA) ～(KEKKA+5)＝計算結果が入る
        MOV     DX,3                  ←かけるバイト数
KAKLP:  MOV     DI,OFFSET ADATA+2
        MOV     AL,[SI]
        CALL    KAKSS
        MOV     AL,[SI]
        SHR     AL,1
        SHR     AL,1
        SHR     AL,1
        SHR     AL,1
        MOV     DI,OFFSET ADA10+3
        CALL    KAKSS
        DEC     SI
        DEC     BX
        DEC     DX
        JNZ     KAKLP
        POP     ES
        POP     DS
        RET
KAKEX   ENDP

KAKSS   PROC                          ←桁ごとのかけ算
        AND     AL,00001111 B
        JE      KAKRT
        MOV     CL,AL
        MOV     CH,0
KAKSL:  PUSH    BX
        PUSH    DI
        MOV     AL,[DI]
        ADD     AL,[BX]
        DAA
        MOV     [BX],AL
        DEC     BX
```

```
              DEC     DI
              MOV     AL,[DI]
              ADC     AL,[BX]
              DAA
              MOV     [BX],AL
              DEC     BX
              DEC     DI
              MOV     AL,[DI]
              ADC     AL,[BX]
              DAA
              MOV     [BX],AL
              DEC     BX
              DEC     DI
              MOV     AL,[DI]
              ADC     AL,[BX]
              DAA
              MOV     [BX],AL
              POP     DI
              POP     BX
              LOOP    KAKSL
KAKRT:        RET
KAKSS         ENDP
```

桁ごとのかけ算といっても、1バイトが2桁分に相当していますから、すべての桁を同じように扱うわけにはいきません。先ほどの例でいうと、普通の筆算では2段目は615となりますが、1段目とは「738+6150」という計算です。つまり、2段目の値は10倍して加算しなければならないわけです。もちろん、3段目は100倍して加算することになりますが、これはメモリを1バイトずらすことで対処できます。しかし、次の段があればやはり10倍した値が必要になります。

そこで、最初にADATA(かけられる数)の値を10倍したものを用意しておくと、かける数の上位4ビットも下位4ビットと同じような計算処理ができます。ただし、10倍したものは4バイトのメモリを使いますから、プログラムは4バイト分の計算(KAKSL内における処理)をしなければなりません。さらに、このプログラムを上位/下位共通に使用するためには、ADATAのほうも

4バイトにしておく必要があります。そのため、ADATAの手前にDUMMYとして1バイト（中身＝0）を確保しているわけです。

プログラム(KAKSS)を共通化せず、下位4ビット用に3バイト分の計算をするプログラムを用意すれば、このDUMMYは不要です。このあたりの判断は、営業部長の父ちゃんにまかせることにします。

問 80 BXレジスタの値をBCDに変換

ピラミッド・パワー……。この謎のベールに包まれたパワーを確認するため、私はクフ王の眠る巨大なピラミッドへとやって来ました。
私は、すでに30年の歳月をかけて、この神秘のパワーを求める物理学的計算式を発見したのです。それは、あらゆるパワーが複雑に、しかもバランスよく絡み合った、まるで生命体のような計算式でした。
しかし、その計算式には1つだけ未知数が存在しているのです。これは、現在の数学では発見されていない数値、たとえていうならi（2乗して－1になる数値）のようなものです。私は、その未知数にPp（ピューピー）と名付け、この計算式を成立させたのです。
では、このPpの正体だけを明かしておきましょう。Ppとは、かけても割っても1になるという数値です。例をあげてみます。

543×Pp＝1　　543÷Pp＝1

つまり、あらゆる数値が1に収束してしまうのです。この未知数Ppの存在を裏付けるため、ピラミッド内部のアチコチでPpの測定をしています。その時、マシン語でBCDの数値と十六進数のBXレジスタの値との高速乗算をしたいのです。
ピラミッド内部より、絶大なるご協力を要請します。

デタラム・ペテム（エジプト）

答

ピラミッドだけでなく、単なる四角錐にも謎のパワーが秘められているそうですが、そのパワーとは一体なんなんでしょうか。卵が腐らないとか、頭がよくなるとか、四角錐にまつわるウワサは真実味にあふれています。

そこに突然登場した、このPp（ピューピー）なる未知数……。どこまで本当で、どこまでウソなのか、まったくわからないところにインチキの魅力がありそうです。

しかし、ここでは『BCDの数値×BX』だけが問題です。高速というからには、問78の方法（足し算のループ）では不満なはず。どうにかして、BXレジスタの値をBCD化して、問79のプログラムが使えるようにしなければなりません。

ここにあるプログラムは、BXレジスタの値をBCDに直し、問79にあるプログラムの「BDATA」に入れるというものです。つまり、このプログラム（HLBCD）をコールしてから問79のプログラムをコールすればいいわけです。

```
HLBIT    DB      0,0,0                 ←ビット別加算データ

HLBCD    PROC
         PUSH    DS
         MOV     AX,CS
         MOV     DS,AX
         XOR     AL,AL
         MOV     SI,OFFSET BDATA+2     ←変換先
         MOV     [SI],AL
         MOV     [SI-1],AL
         MOV     [SI-2],AL
         MOV     BX,OFFSET HLBIT       ←（HLBIT）～（HLBIT+2）の初期化
         MOV     [SI],AL
         INC     BX
         MOV     [BX],AL
         INC     BX
         MOV     BYTE PTR [BX],1
         MOV     CX,16                 ←CX＝変換ビット数（16ビット）
HLBLP:   MOV     BX,OFFSET HLBIT+2
         SHR     DX,1
         JNB     HLB01
         CALL    BTADD                 ←ビット＝1のところだけビット別加算データを加算
HLB01:   MOV     AL,[BX]
         ADD     AL,AL                 ←ビット別加算データを2倍する
         DAA
         MOV     [BX],AL
         DEC     BX
         MOV     AL,[BX]
         ADC     AL,AL
         DAA
         MOV     [BX],AL
         DEC     BX
         MOV     AL,[BX]
         ADC     AL,AL
```

```
        DAA
        MOV     [BX],AL
        LOOP    HLBLP
        POP     DS
        RET
HLBCD   ENDP

BTADD   PROC
        MOV     AL,[BX]                ←ビット別加算データを変換先に加算
        ADD     AL,[SI]
        DAA
        MOV     [SI],AL
        DEC     BX
        MOV     AL,[BX]
        ADC     AL,[SI-1]
        DAA
        MOV     [SI-1],AL
        DEC     BX
        MOV     AL,[BX]
        ADC     AL,[SI-2]
        DAA
        MOV     [SI-2],AL
        MOV     BX,OFFSET HLBIT+2
        RET
BTADD   ENDP
```

プログラムの考え方は、BXレジスタを各ビット別にBCD化して、16ビット分加算していくものです。ビット＝1の場合に加算する値（BCD値）は次のようになります。

ビット 0＝00,00,01	ビット 8 ＝00,02,56
ビット 1＝00,00,02	ビット 9 ＝00,05,12
ビット 2＝00,00,04	ビット 10＝00,10,24
ビット 3＝00,00,08	ビット 11＝00,20,48
ビット 4＝00,00,16	ビット 12＝00,40,96
ビット 5＝00,00,32	ビット 13＝00,81,92
ビット 6＝00,00,64	ビット 14＝01,63,84
ビット 7＝00,01,28	ビット 15＝03,27,68

今回は、これらのビット別データを計算によって求めていますが、メモリにデータとして用意しておくと、「ビット別加算データを2倍する」代わりにそのデータアドレスを変更するだけで済むので、メモリ効率は悪くなりますがスピードはアップします。

また、問47のような考え方で、BXレジスタの値を10000、1000、100、10で減算するように割り、その商をBCD化していくという方法もあります。どちらの方法が処理が速いかは、その時のBXレジスタの値によって変わってきますので、状況に応じながら使い分けてください。

これでPp（ピューピー）が発見できるのであれば、盆と正月と誕生日とクリスマスが一度にやって来るでしょう。ついでに、宝くじにも当たるかもしれません……ネ!?

81 マシン語コード分割

ヒッヒッヒッヒ……。
わたしゃ魔女じゃよ。ホラ、あの黒いとんがり帽子をかぶって、魔法のホウキにまたがって空を飛ぶ魔女じゃ。有名じゃから、知っておるじゃろ。
でも、この魔法のホウキが、実はマイクロコンピュータに制御されていたなんてことは知らんじゃろ。このホウキの秘密は、この長い柄にあるのじゃ。この柄にはいくつかのバージョンがあってな、魔女のランクによって交換されるシステムなのじゃよ。
わたしゃ、ランクの低い三等魔女じゃが、秘かにマシン語を覚えて、この柄のプログラムを変えてしまおうと思っているのじゃ。
そこで質問じゃ。このホウキの柄と刷毛（はけ）の関係のように、マシン語コードを分割して使うことがあるそうなのじゃが、そんな魔法のような使い方ができるのじゃろか、というのが質問なのじゃ。
魔女だからといって、イジワルはしないでおくれ……。

西洋の魔女（ノートルダム）

答

確か、ノートルダムに住んでいるのは‘せむし男’のはずでは……？　それにしても、現代の魔女はマシン語まで勉強しなければならないとは、本当にご苦労なこととしか言いようがありません。

質問があまりにも魔女的なので、もう少し普通の人間にもわかるように、問題を具体化してみましょう。

あるマシン語コードがあったとして、そのコードをうまく分割して別なマシン語コードとして使うというわけですが。確かに方法としてはあります。

といっても、むやみに分割しても意味がありません。例えば、条件によってAXレジスタに0または0以外の数値を格納するという場面にしばしば出会うことがありますが、これをプログラムすると次のようになります。

```
        ⋮
    INT     21H
    JNB     CODE2
```

```
CODE1:    MOV    AX,0FFFFH
          JMP    LB001
CODE2:    XOR    AX,AX
LB001:      ⋮
            ⋮
```

このような場合「MOV　AX,0FFFFH」のオペランドである$0FFFF_H$が、特に0以外のどのような数値でもよいのであれば、このオペランド部分を「XOR　AX,AX」のマシン語コードである$0C031_H$とすることによって次のようなプログラムが組めるわけです。

```
            ⋮
          INT    21H
          JNB    $+3
          MOV    AX,0C031H
            ⋮
```

これは、キャリーフラグの状態によって、「MOV　AX,0C031H→AX=C031H」または、「XOR　AX,AX → AX=0」となり、プログラムの目的は果たせることになります。しかも4バイトのメモリの節約になるのです。

ほかにも、色々あるでしょうが、それは魔女さんへの課題としておきます。うまくいけば、三等魔女のホウキも一挙に一等魔女のホウキに化けられるかもしれません。

82 BCDどうしの割り算（割る数の桁数固定）

鉄人86号がいいだろうか、それとも鉄腕インテルがいいかな、やっぱりハチロクマンにしようか……。
ネーミングが決まらないから、なかなか設計にかかれない。人間が月へ行く時代だというのに、いまだに本物のロボットが作れないというのは、はなはだ遺憾である。自動車メーカーには、部分的にロボットらしきものがあるが、いつまで待ってもあれは歩かないようだ。
マ、順序としては鉄人86号から完成させるのが筋だろうな。もちろん、ICは8086系を使うことになる。では、さっそく設計にかかろう。
……ウ～ム、電波を受けてから手を動かすためには、高速な割り算をしなければならないのか。一応、割る数は8888に固定しよう。割られる数は不定だが、8桁だな。ということは、BCDによる筆算タイプの割り算でないと無理だ。
しかし、減算ループによる割り算ならできそうだが、筆算のようにするには難しそうだぞ。設計の前に教えてもらわなければなるまい。
鉄人のために、よろしく頼む。

敷島博士（山梨）

答

鉄人86号……。名前はカッコイイけど、完成した鉄人が活躍する場所を捜すのに苦労しそうです。

まず、相手がいない。今どき、わざわざ目立つロボットを作って悪いことをしようなどという悪人はいないでしょう。おまけに、下手に歩けば道路はこわすし電線にも引っかかる。それではとロケットが火を噴けば、周りは火事になって大惨事……。

これでは、せっかくの夢がこわれそうです。余計なことを考えずに、BCDの筆算タイプの割り算を実現することにしましょう。

かけ算の時もそうでしたが、BCDでは割り算を二進数的な考え方で実行することはできません。そのため、商の桁ごとに減算のループによる割り算を行うことになります。

今回は、割られる数が8桁、割る数が8888に固定されているので、4バイト

のBCD数値を2バイト（4桁）のBCD数値で割るという計算ですが、割られる数、割る数の桁数は割り算の重要なポイントです。

商の桁数＝8－4＋1

また、BCDの計算では、その桁の商がメモリの上位4ビットに入るか、下位4ビットに入るかも区別しなければなりません。今回の場合は、商が5桁ですからメモリの下位4ビットから入れることになります。

ここではサンプルデータとして割られる数＝10967792としていますが、もし割り切れない数となった場合は小数点以下切り捨てです。ただし、小数点はユーザー側の区切りですから、ダミーのメモリ（中身＝0）を追加すれば、小数点以下何位まででも求めることができます。その場合、割る数（BDATA）や商（KEKKA）のメモリも計算に合わせて追加しなければなりません。

割り算の最後は常に切り捨てですから、四捨五入したい場合は問77の方法で自由にしてください。

```
KEKKA    DB      0,0,0                  ←商が入る
ADATA    DB      10H,96H,77H,92H        ←割られる数（8桁）
BDATA    DB      88H,88H,0,0            ←割る数（上位4桁）
                                          下4桁は桁合わせのためのダミー

WARIX    PROC
         PUSH    DS
         MOV     AX,CS
         MOV     DS,AX
         MOV     SI,OFFSET KEKKA
         MOV     CL,0                   ←CL＝桁ごとの商
         CALL    WARI1                  ←下位4ビットの商を求める
         CALL    WARI2                  ←上位4ビット/下位4ビットの商を求める
         CALL    WARI2                  ←上位4ビット/下位4ビットの商を求める
         POP     DS
         RET
WARIX    ENDP

WARI2    PROC
         MOV     CL,0
         CALL    WAXCC
```

```
        SHL     CL,1                        ←商を上位4ビットに移す
        SHL     CL,1
        SHL     CL,1
        SHL     CL,1
        CALL    WARI1
        RET
WARI2   ENDP

WARI1   PROC
        CALL    WAXCC
        MOV     [SI],CL
        INC     SI
        RET
WARI1   ENDP

WAXCC   PROC
        MOV     DI,OFFSET ADATA+3    ①     ←桁ごとの商を求める割り算
        MOV     BX,OFFSET BDATA+3    ②
        CLC
WACL0:  MOV     CH,4                 ③     ←CH=BCD数値のバイト数
        PUSH    DI                          ←減算ループによる割り算
        PUSH    BX
SBCL0:  MOV     AL,[DI]
        SBB     AL,[BX]
        DAS
        MOV     [DI],AL
        DEC     DI
        DEC     BX
        DEC     CH
        JNZ     SBCL0
        POP     BX
        POP     DI
        JB      WAED0
        INC     CL
        JMP     WACL0
WAED0:  MOV     CH,4                 ③     ←減算しすぎた分を加算する
        CLC
ADCL0:  MOV     AL,[DI]
        ADC     AL,[BX]
        DAA
```

```
        MOV     [DI],AL
        DEC     DI
        DEC     BX
        DEC     CH
        JNZ     ADCL0
        XOR     AL,AL
        MOV     DX,4              ③ ←割る数を次の桁用に右シフトする
        PUSH    CX
        MOV     CL,4
HLRRD:  INC     BX
        MOV     AH,[BX]
        ROR     AX,CL
        ROR     AL,CL
        MOV     [BX],AH
        DEC     DX
        JNZ     HLRRD
        POP     CX
        RET
WAXCC   ENDP
```

なお、プログラムを実行すると、割られる数(ADATA)は割り算の余りとなり、割る数(BDATA)は破壊されます。必要があれば、退避してから実行してください。

そういえば、敷島博士の写真が同封されていましたが、年老いて、博士というよりまるで仙人のような風貌でした……。

83 内部割り込みと外部割り込み

ここをどこだと思いますか？　咲き乱れる美しい花。夢と理想と現実が完全に一致した憧れの都……。そうです、ここは天国です。下界でも、『天国よいとこ一度はおいで。酒はウマイし、ねェちゃんはキレイだ』と歌われていますネ。
とはいえ、ここは一度来たら二度と下界へは帰れませんから注意してください。下界に未練があるうちは決して来てはなりません。なーに、無理しなくても誰でもいつかは来られますから、あせることはないのです。
もちろん、ここではパソコンを楽しむことも自由自在です。プログラムだって好きなだけできます。実は、私は下界では割り込みが得意でしたが、ここに来て初めて8086を使うようになったのです。ところが、8086には外部割り込みと内部割り込みがあるというじゃないですか。これまでは外部割り込みだけでしたから、内部割り込みがどういうものなのか、ピンと来ないのです。どうぞ、よろしく。
ア、返事はテレパシーで送ってください。テレ・ナンバーは(03)16000-9801です。

天馬天之介（天国）

答

なんとなく、こわ～いような気もする質問ですが、これ普通の手紙で来たんですヨ。しかも、封筒の裏側にはちゃんと住所が……。アレ？　よく見れば、『キャバレー天国』なんて書いてあるじゃないですか。驚かさないでください。私は、いたって気が小さいんですからネ。

さて、他のCPUで外部割り込みを使ったことがあれば、ユーザーが使う内部割り込みは非常に簡単です。そもそも、内部割り込みは「ソフトウェア割り込み」とも言って、プログラム側から割り込みを発生させるものなのです。つまり、プロシージャ感覚でコールして使うのです。

ただ、CALL命令がアドレスでコールするのに対して、内部割り込みはINT命令で割り込み番号を設定してコールします。

◆プロシージャコール
```
CALL    PROC1    ：アドレス指定でコールする
```
◆内部割り込みの例
```
INT    21H        ：割り込み番号でコールする
```

内部割り込みには、ユーザーがINT命令で発生させるほかに、CPUが使うために予約しているもの、MS-DOSなどのシステム側で予約しているもの、8086を搭載した機種で予約しているものなどがあります。

したがって、ユーザーが内部割り込みを使う場合には、ユーザー用に解放されている割り込みを使うことになります。例えば、PC-9801などでは、「INT 40H」~「INT 7FH」となります。

ここで、INT命令の動作を確認してみましょう。

1・・・フラグをスタックへプッシュする
2・・・割り込みフラグとトラップフラグをクリアする
3・・・コードセグメント（CS）をスタックへプッシュする
4・・・CSへ割り込み番号に対応するセグメントアドレスをベクターテーブルからロードする
5・・・インストラクション・ポインター（IP）をスタックへプッシュする
6・・・IPへ割り込み番号に対応するオフセットアドレスをベクターテーブルからロードする

割り込みプロシージャを組む時には、フラグがプッシュされた後に、インタープトフラグがクリアされることに注意してください。なお、割り込みプロシージャに対するリターン命令は「IRET」が使われます。

ユーザーの作った割り込みプロシージャは何らかの方法で、空いている割り込みテーブルにエントリーを登録しなければなりませんが、MS-DOSには割り込みプロシージャのエントリーを登録するファンクションが用意されています。通常はこれを利用して登録するといいでしょう。ただし、割り込み処理は機種に依存する部分が多いので、機種の特性を十分に把握した上でプログラムを組んでください。

それでは、このテレパシーが『キャバレー天国』に届くことを祈っております。

84 MAKEの利用

えいっ……!!
たあー……!!
うーやーたあーっ!!
求道者の道はつらい。常に正しい道を求めなければならない。しかも道は自分で切り開かねばならない。おっと……いけない、つい弱音をはいてしまった。えっ？　何の求道者かって、もちろん、マシン語道だ、拙者はマシン語歴１年、自称、マシン語道迷人位の位を持っておる。ところで、困ったことに、MS-DOSの外部コマンドにMAKEを発見してしまった。マシン語開発にはとても便利であるそうだが、どうも不安でしょうがない、なにが不安かって、何が不安かわからないから困っておるのだ。求道者としてはMAKEを使うべきであることはわかっている。そこで相談だが、この不安を取り除いてはくれないだろうか？
そうしたら、秘伝中の秘伝をさずけてやろう。信用されないと困るので、少しだけ内容を公開してやる。その秘伝とは、≪意志を持った無だ≫。どうだ、まいったろう。
どういうことかって、そこまでは公開できない。後は、この不安を取り除いてくれてからの話だ。では、よろしくたのむ!!

マシン語迷人（人間界）

答

秘伝中の秘伝を公開してくれるとなると、不安を取り除いてあげなければならないでしょう。でも、内容がナイヨウなので不安ですが……。

さて、MAKEを発見したとなるとマシン語の腕前もかなりのもの、プログラム開発にも余裕が出てきたところかもしれません。MAKEは、次のようにすれば起動されます。

```
A>MAKE <メイクファイル名>
```

もちろん、MAKEを起動する場合、「MAKE.EXE」という実行ファイルが必要となります。さらに、MAKEを実行する手順を示したメイクファイルを作らなければなりません。

このメイクファイルは「CONFIG.SYS」と同じように、テキストファイルで

すから、エドリン（EDLIN）などのエディタを使って、決められた書式で記述し作成します。なお、メイクファイル名は、開発プログラムと同じ名前を、拡張子を付けないで用いるのが慣習となっているようです。

MAKEで最も重要なのが、このメイクファイルです。メイクファイルの書式は次の通りですから、参考にしてください。

目的ファイル、関連ファイル、そしてコマンドを組み合わせて1つの作業記述単位となります。この作業記述は1つのメイクファイル中、いくつ入れてもかまいませんが、作業記述と作業記述の間は、最低1行は、空けなければなりません。また、各ファイルが、メイクファイルと同じディレクトリにない場合には、パス名が必要になります。

MAKEは、目的ファイルの作成日時以後に、各関連ファイルに修正があればコマンドを自動的に実行するようになっています。また、目的ファイルが存在しない場合にもコマンドを実行します。もし、目的ファイルが存在し、かつ関連ファイルに修正がなければコマンドは実行されません。これだけでは不安を解消できそうにもありませんから、実際のメイクファイルの例を示しておきましょう。

サンプル－1

```
TEST1.OBJ  : TEST1.ASM
        MASM   TEST1;

TEST1.EXE  : TEST1.OBJ
        LINK   TEST1;
```

サンプル－2

```
TEST1.OBJ  : TEST1.ASM TEST1-2.ASM TEST1-3.ASM TEST1-4.ASM
        MASM   TEST1;

TOOL1.OBJ  : TOOL1.ASM
        MASM   TOOL1;

TOOL2.OBJ  : TOOL2.ASM
        MASM   TOOL2;

TEST1.EXE  : TEST1.OBJ TOOL1.OBJ TOOL2.OBJ
        LINK TEST1 TOOL1 TOOL2,,/MAP;
        MAPSYM TEST1
```

これで不安を解消できたでしょうか？　もっとも、一番の方法は知識を身に付けるのではなく、実際に使ってみることですが……。

85 BCDに関するミニ・テクニック

人間はコンピュータを作った。そして、それは人工知能へと発展しつつある。最近ではファジィ理論という、人間らしい曖昧さを求めた研究も盛んに行われているというではないか。つまり、人間は自分と同じことをコンピュータにさせることで、生命を創造した神になろうとしているのだ。
……だが、おいらの存在を忘れてもらっちゃ困るぜ。コホン!! おいらはコンピュータなどには絶対にマネのできない第三の知能さ。おいらの正体はまだ発見されていないようだが、誰でもおいらの存在は知っているんだ。
おいらの名は『夢』……。夢を司る脳に住んでいるのさ。いくらコンピュータが人間のマネをしたところで、夢は見れまい。そこに神の偉大さがあるのだ。
だけど、おいらにも夢がある。それはおいらの存在をコンピュータ化してもらうことなんだ。そのためには、BCDどうしの比較やゼロチェックが自由自在に、しかもスバヤクできなければならない。
人工夢脳のために、こういったBCDのミニ・テクを公開する気はないだろうか。もっとも、それは人工夢脳へのスタートに過ぎないが……。

夢見る夢（夢脳）

答

近くて遠い夢の国……。

行けそうで行けない夢の国……。

現実のようでどこかが違う夢の国……。

夢をプログラムで実現できるかどうかはわかりませんが、その前に夢を録画するビデオを開発してほしいものです。見ている時は真実そのもの、しかし実際にはデタラメで支離滅裂でウソと幻に包まれたナゾの世界……。そのナゾの解明に役立てるなら、BCDのミニ・テクなどいくらでも公開しましょう。

(1) BCDどうしの比較

基本的には減算をすればいいのですが、下位桁から減算をしていくと全メモリを減算しないと比較できません。人間的な思考法では、上位桁から比較をするほうが自然です。このプログラムは「CMP DI（BCDのアドレス）,SI（BCDのアドレス）」を実行し、ゼロ／キャリーフラグで判定を示すもので

す。例えば「CMP ①,②」をしたければ、DIレジスタ=OFFSET ADATA 、SIレジスタ=OFFSET BDATAとして、AXレジスタにデータ・セグメント値を格納してCPBCDをコールします。

```
ADATA  DB     12H,34H,56H,78H,90H,12H   ←①
BDATA  BD     11H,22H,33H,44H,55H,66H   ←②

CPBCD  PROC
       PUSH   DS
       PUSH   ES
       MOV    DS,AX
       MOV    ES,AX
       MOV    CX,6                      ← CX=BCDデータのバイト数
       CLD
       REPZ   CMPSB
       POP    ES
       POP    DS
       RET
CPBCD  ENDP
```

(2) BCDのゼロチェック

すべての桁（メモリ）がゼロかどうかを調べるわけですが、これも上位から調べるほうが人間的でしょう。DIレジスタに調べたいBCDのオフセットアドレスを入れ、AXレジスタにセグメント・アドレスをセットしてコールしてください。

```
BCDZF  PROC          ← AXにデータ・セグメント値、DIにオフセット・アドレスを格納してコール
       PUSH   ES
       MOV    ES , AX
       XOR    AX , AX
       MOV    CX,6   ← BCDデータのバイト数
       CLD
       REPZ   SCASB
       POP    ES
       RET
BCDZF  ENDP
```

(3) BCDの符号について

BCDの数の中には符号を含むことができませんから、符号が必要な場合には符号用のメモリを1バイト用意しなければなりません。例えば、0ならプラス、1ならマイナスと決めておいて、加減算の場合にはその符号と照らし合わせてから実行するようにするのです。

ただし、計算を加減算だけに限定すれば、符号付き数値のように扱うこともできます。例えば、2桁のBCD数値で15とあれば、85を−15と考えてもいいわけです。

20−15＝05

20＋85＝05 ← 100以上は計算されない

このような符号変換は、いわばBCD版NEG命令ということになります。

次のプログラムは3バイト用のBCD版NEG命令です。AXレジスタにデータ・セグメント・アドレス、DIレジスタにBCD数値の最下位アドレスを入れてコールしてください。

```
BCNEG   PROC
        PUSH  DS
        CLC
        MOV   DS,AX
        MOV   CX,6          ←BCDデータのバイト数
BNGLP:  MOV   AL,0
        SBB   AL,[DI]
        DAS
        MOV   [DI],AL
        DEC   DI
        LOOP  BNGLP
        POP   DS
        RET
BCNEG   ENDP
```

いずれも大したテクニックではありませんが、BCD数値を自在に扱うための基礎として覚えておくと便利です。その程度ですから、これが夢の人工夢脳の開発に役に立つことは夢にも思えません……。

86 テーブル処理で複雑な計算を

放浪……。そこには未知のものに出会える夢がある。しかし、最近はテレビが発達しすぎたせいで、夢が減ってしまった。
昔は『知床旅情』なんていう歌は、北海道へ旅した者が覚えてくる歌だったのだ。だから、この歌がテレビを通じて流行ったとたん、夢が1つ消えた。
かつて、私は日本最北端の島、礼文島を旅した。ユースホステルでは、元フォークグループの若者が歌を教えていた。それは、『旅の終わり』、『島を愛す』という2つの歌だった。これらは、少なくともメジャーな歌にはならなかった。
もしかすると、この歌はもう誰も知らないかもしれない……。私は、ときどき口ずさんでは、たった一人の優越感に浸っている。
私は、さすらいのプログラマー……。もちろん、かけ算は知っている。実は X^2 を多用するプログラムがあるのだが、毎回計算しているので時間がかかる。きっと、もっといい方法があると思うのだが、現状ではわからない。
しかし、マシン語には未知のテクニックを発見できる夢がある。まるで放浪の旅をしているようだ。私は、そんなマシン語が好きだ。

さすらい人（礼文島）

答

テレビというのは、夢を与えてくれているようで、実は夢を奪っているのです。なぜなら、夢は想像の中から生まれるものだから……。

マシン語の世界は限りない想像の世界です。そこには、高級言語では味わうことのできない未知の魅力と無限の可能性があります。もしかすると、最も身近な未踏の秘境かもしれません。

……が、マシン語で秘境探検をするには発想の転換が必要です。この X^2 も、多用するからにはかけ算処理を省きたいものです。次のプログラムは、ALレジスタ（0～255）を2乗し、その結果をDXレジスタに求めるというものです。

プログラム自体は、問27でジャンプ・テーブルとして用いたものと似ています。しかし、アイデアは似ていても内容はまったく違います。さらに、この X^2 という計算も見本の1つにすぎません。この用法の本当の価値は、複雑な計

```
KDATA   DW      0,1,4,9,16,25              ←0～255を2乗した数値を
        DW      36,49,64,81,100              データとして用意しておく
                 ⋮      ⋮
        DW      63504,64009,64516,65025

XTIMX   PROC                               ←AL²の結果をDXに求める
        MOV     AH,0
        SHL     AX,1
        MOV     SI,OFFSET KDATA
        ADD     SI,AX
        MOV     DX,[SI]
        RET
XTIMX   ENDP
```

(注) DS＝CSと仮定する

算をさせた場合に現れてくるのです。

また、計算結果を直接BCDにすることもできます。つまり、テーブルの内容に工夫を凝らせば、いくらでも応用の範囲は広がるということです。人跡未踏のテクニックというわけではありませんが、マシン語のちょっとした秘境といえるでしょう。

しかし、元の値がBCDの場合は、簡単にテーブルを利用できません。テーブルは十六進数のアドレスで示されますから、BCDのままでは虫喰いテーブル(A～Fのあるアドレスを使用しないテーブル)となってしまうからです。そこで、参考までに0～65535のBCD数値を十六進数に変換し、BXレジスタに入れるプログラムを載せておきます。

```
BCDDT   DB      06H,55H,35H               ←BCDの数値

BCDHL   PROC
        MOV     SI,OFFSET BCDDT
        XOR     BX,BX
        MOV     AL,[SI]
        MOV     CX,10000
        CALL    KAI4B                     ←10000の位の計算
        MOV     AL,[SI+1]
```

```
        MOV   CX,1000
        CALL  JOI4B          ←1000の位の計算
        MOV   CX,100
        CALL  KAI4B          ←100の位の計算
        MOV   AL,[SI+2]
        MOV   CX,10
        CALL  JOI4B          ←10の位の計算
        AND   00001111B
        MOV   CL,AL
        ADD   BX,CX          ←1の位の計算
        RET
BCDHL   ENDP

JOI4B   PROC
        PUSH  AX             ←上位4ビットの計算
        SHR   AL,1
        SHR   AL,1
        SHR   AL,1
        SHR   AL,1
        CALL  KAI4B
        POP   AX
        RET
JOI4B   ENDP

KAI4B   PROC
        AND   AL,00001111B   ←下位4ビットの計算
        CBW
        MUL   CX
        ADD   BX,AX
        RET
KAI4B   ENDP
```

（注）DS＝CSと仮定する

このプログラムを利用すれば、例えば「0～99のBCD数値」→「計算結果＝6バイト（12桁）のBCD数値」などといったテーブルも簡単に実現できます。もちろん、テーブルのために使用するだけでなく、単なる変換ルーチンとしてBCD数値のままでは不便という場合に活用してもかまいません。

テーブルというのは応用範囲が大変広いテクニックですから、プログラムが

複雑になったりダラダラと長くなりそうなときは、テーブル化を検討する価値があります。もしかすると、まったく新しい用法や秘術に出会えるかもしれません。

『新テクの陰にテーブルあり』

これも、マシン語の極意の1つといえるでしょう。マシン語にはこのようにメジャーなテクニックもあれば、日陰の鈴蘭のようなマイナーなテクニックもあります。私も、そんなマシン語が好きです……。

三の章　正しいマシン語のために

1 ……… ニモニック表のオペランドで使われている記号の意味

2 ……… 8086のレジスタ紹介

3 ……… フラグ記号の意味

4 ……… フラグの名称

5 ……… クロック記号の意味

6 ……… オペレーションコード・フィールド

7 ……… 8または16ビット汎用レジスタの選択

8 ……… セグメント・レジスタの選択

9 ……… メモリ・アドレッシング

10 ……… 8086ニモニック一覧表（機能別アルファベット順）

- 10-1 …… 加算命令
- 10-2 …… 減算命令
- 10-3 …… 乗算命令
- 10-4 …… 除算命令
- 10-5 …… データ転送命令
- 10-6 …… 論理演算命令
- 10-7 …… 分岐命令
- 10-8 …… ストリング命令
- 10-9 …… ストリング・プリフィックス命令
- 10-10 …… フラグ制御命令
- 10-11 …… CPU制御命令
- 10-12 …… セグメント・オーバーライド命令

〈1〉ニモニックのオペランドで使われている記号の意味

オペランド	意　味
reg	8または16ビットの汎用レジスタ
reg 8	8ビットの汎用レジスタ
reg 16	16ビットの汎用レジスタ
mem	8または16ビット・メモリロケーション
mem 8	8ビット・メモリロケーション
mem 16	16ビット・メモリロケーション
mem 32	32ビット・メモリロケーション
acc	AX、ALレジスタ
sreg	セグメントレジスタ
imm	8ビットまたは16ビットの数値
imm 8	8ビットの数値
imm 16	16ビットの数値
nearproc	現在の命令が置かれているコードセグメント内のプロシージャ
farproc	別のコードセグメント内のプロシージャ
nlabel	命令が置かれているコードセグメント内のラベル
flabel	別のコードセグメント内のラベル
slabel	命令の終わりから－128～＋127バイトの範囲のラベル
memptr 16	制御が移されようとしているオフセットが格納してあるワード
memptr 32	制御が移されようとしているオフセットとセグメントが格納してあるダブルワード
regptr 16	制御が移されようとしているオフセットが格納してあるレジスタ
pvalue	スタックからPOPされるバイト数（偶数）
exop	コプロセッサの命令中にエンコードされる数値　(0～63)

〈2〉 8086のレジスタ紹介

名　前	説　明
AX	アキュムレータ（16ビット）
AL	AXの下位8ビット（アキュムレータ8ビット）
AH	AXの上位8ビット
BX	ベース・レジスタ（16ビット）
BL	BXの下位8ビット
BH	BXの上位8ビット
CX	カウンタレジスタ（16ビット）
CL	CXの下位8ビット
CH	CXの上位8ビット
DX	データ・レジスタ（16ビット）
DL	DXの下位8ビット
DH	DXの上位8ビット
SI	ソース・インデックス・レジスタ（16ビット）
DI	ディスティネーション・インデックス・レジスタ(16ビット)
CS	コードセグメント・レジスタ（16ビット）
DS	データセグメント・レジスタ（16ビット）
ES	エクストラセグメント・レジスタ（16ビット）
SS	スタックセグメント・レジスタ（16ビット）
SP	スタックポインタ（16ビット）
BP	ベースポインタ（16ビット）
IP	インストラクションポインタ（16ビット）
F	フラグ・レジスタ（16ビット）

〈3〉 フラグ記号の意味

記号	意味
・	変化なし
?	不定
X	結果に従って変化する
0	リセット
1	セット
r	退避した値をストアする

〈4〉 フラグの名称

AF	補助キャリーフラグ	DF	ディレクションフラグ
CF	キャリーフラグ	IF	インターラプトフラグ
PF	パリティーフラグ	OF	オーバーフローフラグ
SF	サインフラグ	TF	トラップフラグ
ZF	ゼロフラグ		

〈5〉 クロック記号の意味

記 号	意 味
N	N回かけあわせる
/	A/B (AまたはB)
−	A−B (AからB)
+EA	●ダイレクト16ビット・オフセット・アドレス (6) ●ベースまたはインデックス・レジスタによるインダイレクト (5) ●インデックス・レジスタとベース・レジスタとの和によるインダイレクト (7 or 8) ●ディスプレイスメントを伴ったベースまたはインデックス・レジスタによるインダイレクト (9) ●ディスプレイスメントを伴ったインデックス・レジスタとベース・レジスタとの和によるインダイレクト (11 or 12) (注) 奇数アドレスに対しては4クロック加える。またセグメント・オーバーライドにはさらに2クロック加える

〈6〉 オペレーションコード・フィールド

名　前	説　明
W	ワード／バイト・フィールド（0 or 1）
reg	レジスタ・フィールド（000～111）
sreg	セグメント・レジスタ・フィールド（00～11）
r/m	レジスタ／メモリ・フィールド（000～111）
mod	モード・フィールド（00～10）
S:W	S:W＝01のときdata＝16ビット、それ以外はdata＝8ビット S:W＝11のときバイトデータのサインが拡張されて16ビット・オペランドを作る
XXX	ESCオペレーションコードのはじめの3ビット
YYY	ESCオペレーションコードの2番目の3ビット

〈7〉 8または16ビット汎用レジスタの選択

* reg or r/m	W=0	W=1
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

＊ r/mはmodのない場合

〈8〉 セグメント・レジスタの選択

sreg	内容
00	ES
01	CS
10	SS
11	DS

〈9〉 メモリ・アドレッシング

r/m \ mod	00	01	10
000	BX+SI	BX+SI+disp 8	BX+SI+disp 16
001	BX+DI	BX+DI+disp 8	BX+DI+disp 16
010	BP+SI	BP+SI+disp 8	BP+SI+disp 16
011	BP+DI	BP+DI+disp 8	BP+DI+disp 16
100	SI	SI+disp 8	SI+disp 16
101	DI	DI+disp 8	DI+disp 16
110	DIRECT ADDRESS	BP+disp 8	BP+disp 16
111	BX	BX+disp 8	BX+disp 16

〈10〉 8086ニモニック一覧表

[10-1] 加算命令（アルファベット順）

ニモニック	オペランド	第1バイト HEX	第1バイト 7 6 5 4 3 2 1 0	第2バイト HEX	第2バイト 7 6 5 4 3 2 1 0
AAA		37	0 0 1 1 0 1 1 1		
ADC	reg,reg mem,reg reg,mem reg,imm mem,imm acc,imm		0 0 0 1 0 0 0 W 0 0 0 1 0 0 0 W 0 0 0 1 0 0 1 W 1 0 0 0 0 0 S W 1 0 0 0 0 0 S W 0 0 0 1 0 1 0 W		1 1 reg r/m mod reg r/m mod reg r/m 1 1 0 1 0 r/m mod 0 1 0 r/m
ADD	reg,reg mem,reg reg,mem reg,imm mem,imm acc,imm		0 0 0 0 0 0 0 W 0 0 0 0 0 0 0 W 0 0 0 0 0 0 1 W 1 0 0 0 0 0 S W 1 0 0 0 0 0 S W 0 0 0 0 0 1 0 W		1 1 reg r/m mod reg r/m mod reg r/m 1 1 0 0 0 r/m mod 0 0 0 r/m
DAA		27	0 0 1 0 0 1 1 1		
INC	reg 8 mem reg 16	FE	1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 W 0 1 0 0 0 reg		1 1 0 0 0 r/m mod 0 0 0 r/m

バイト数	クロック数	フラグ ODISZAPC	内　容
1	4	?・・??X?X	Ascii Adjust for Addition 十進アスキーコード間における加算結果をALレジスタに求めたとして、その補正を行う場合に使われる 【使用例】 MOV AH,00 H : AH ← 00_H MOV AL, 35 H : AL ← 35_H ADD AL, 35 H : AL ← $6A_H$ AAA : AX ← 0100_H
2 2−4 2−4 3−4 3−6 2−3	3 16+EA 9+EA 4 17+EA 4	X・・XXXXX X・・XXXXX X・・XXXXX X・・XXXXX X・・XXXXX X・・XXXXX	ADd with Carry キャリーを含む加算を行う 【使用例】 ADC AX,BX : AX←AX+BX+CF
2 2−4 2−4 3−4 3−6 2−3	3 16+EA 9+EA 4 17+EA 4	X・・XXXXX X・・XXXXX X・・XXXXX X・・XXXXX X・・XXXXX X・・XXXXX	ADDition　加算命令 【使用例】 ADD AX,BX : AX←AX+BX
1	4	?・・XXXXX	Decimal Adjust for Addition 二進化十進数における加算結果をレジスタALに求めたとして、その補正をする 【使用例】 MOV AL,35 H : AL← 35_H ADD AL,35 H : AL← $6A_H$ DAA : AL← 70_H
2 2−4 1	3 15+EA 2	X・・XXXX・ X・・XXXX・ X・・XXXX・	INCrement by 1 オペランドの内容を+1する 【使用例】 INC AX

[10-2] 減算命令

ニモニック	オペランド	第1バイト HEX	第1バイト 7 6 5 4 3 2 1 0	第2バイト HEX	第2バイト 7 6 5 4 3 2 1 0
AAS		3F	0 0 1 1 1 1 1 1		
CMP	reg,reg mem,reg reg,mem reg,imm mem,imm acc,imm		0 0 1 1 1 0 0 W 0 0 1 1 1 0 0 W 0 0 1 1 1 0 1 W 1 0 0 0 0 0 S W 1 0 0 0 0 0 S W 0 0 1 1 1 1 0 W		1 1 reg r/m mod reg r/m mod reg r/m 1 1 1 1 1 r/m mod 1 1 1 r/m
DAS		2F	0 0 1 0 1 1 1 1		
DEC	reg 8 mem reg 16	FE	1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 W 0 1 0 0 1 reg		1 1 0 0 1 r/m mod 0 0 1 r/m
NEG	reg mem		1 1 1 1 0 1 1 W 1 1 1 1 1 1 1 W		1 1 0 1 1 r/m mod 0 1 1 r/m
SBB	reg,reg mem,reg reg,mem reg,imm mem,imm acc,imm		0 0 0 1 1 0 0 W 0 0 0 1 1 0 0 W 0 0 0 1 1 0 1 W 1 0 0 0 0 0 S W 1 0 0 0 0 0 S W 0 0 0 1 1 1 0 W		1 1 reg r/m mod reg r/m mod reg r/m 1 1 0 1 1 r/m mod 0 1 1 r/m
SUB	reg,reg mem,reg reg,mem reg,imm mem,imm acc,imm		0 0 1 0 1 0 0 W 0 0 1 0 1 0 0 W 0 0 1 0 1 0 1 W 1 0 0 0 0 0 S W 1 0 0 0 0 0 S W 0 0 1 0 1 1 0 W		1 1 reg r/m mod reg r/m mod reg r/m 1 1 1 0 1 r/m mod 1 0 1 r/m

バイト数	クロック数	フラグ ODISZAPC	内　容
1	4	?・・??X?X	Ascii Adjust for Subtraction 十進アスキーコード間の減算結果をALレジスタに求めたとして、その補正をする 【使用例】 MOV AH,10 H : AH ← 10_H MOV AL, 35 H : AL ← 35_H SUB AL, 36 H : AL ← $0FF_H$ AAS : AX ← $0F09_H$
2 2－4 2－4 3－4 3－6 2－3	3 9＋EA 9＋EA 4 10＋EA 4	X・・XXXXX X・・XXXXX X・・XXXXX X・・XXXXX X・・XXXXX X・・XXXXX	CoMPare destination to source 比較 【使用例】 CMP AX,BX JNE ＊＊＊＊
1	4	?・・XXXXX	Decimal Adjust for Subtraction 二進化十進数における減算結果をレジスタALに求めたとして、その補正をする 【使用例】 MOV AL,35 H : AL← 35_H SUB AL,36 H : AL← $0FF_H$ DAS : AL← 99_H
2 2－4 1	3 15＋EA 2	X・・XXXX・ X・・XXXX・ X・・XXXX・	DECrement by 1 オペランドの内容を－1する 【使用例】 DEC AX
2 2－4	3 16＋EA	X・・XXXXX X・・XXXXX	NEGate レジスタやメモリの内容の補数 【使用例】 NEG AX
2 2－4 2－4 3－4 3－6 2－3	3 16＋EA 9＋EA 4 17＋EA 4	X・・XXXXX X・・XXXXX X・・XXXXX X・・XXXXX X・・XXXXX X・・XXXXX	SuBtract with Borrow キャリーを含む減算を行う 【使用例】 SBB AX,BX : AX←AX－BX－CF
2 2－4 2－4 3－4 3－6 2－3	3 16＋EA 9＋EA 4 17＋EA 4	X・・XXXXX X・・XXXXX X・・XXXXX X・・XXXXX X・・XXXXX X・・XXXXX	SUBtraction 減算命令 【使用例】 SUB AX,BX : AX←AX－BX

[10-3] 乗算命令

ニモニック	オペランド	第1バイト HEX	第1バイト 76543210	第2バイト HEX	第2バイト 76543210
AAM		D4	11010100	0A	00001010
IMUL	reg 8	F6	11110110		11101 r/m
	mem 8	F6	11110110		mod 101 r/m
	reg 16	F7	11110111		11101 r/m
	mem 16	F7	11110111		mod 101 r/m
MUL	reg 8	F6	11110110		11100 r/m
	mem 8	F6	11110110		mod 100 r/m
	reg 16	F7	11110111		11100 r/m
	mem 16	F7	11110111		mod 100 r/m

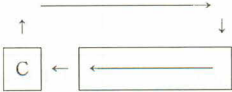
バイト数	クロック数	フラグ ODISZAPC	内　容
2	83	?・・XX?X?	Ascii Adjust for Multiplication 十進アスキーコード間における乗算結果をALレジスタに求めたとして、その補正を行う場合に使われる。乗算命令にはアスキーコード数どうしの命令は用意されていないので、上位4ビットはあらかじめ0クリアしておかなければならない 【動作】 AH←ALを10で割った商 AL←ALを10で割った余り 【使用例】　3×5をアスキーコードで MOV AL, 33 H　：3のアスキーコード MOV BL, 35 H　：5のアスキーコード AND AL, 0FH　：AL←03_H AND BL, 0FH　：BL←05_H MUL BL　：AL←$0F_H$ AAM　：AX←0105_H OR AX,3030 H　：AX←3135_H
2 2−4 2 2−4	80−98 (86−104)+EA 128−154 (134−160)+EA	X・・????X X・・????X X・・????X X・・????X	Integer MULtiplication 符号付きの乗算命令 1バイトどうしの乗算と2バイトどうしの乗算が可能 1バイト乗算の時にはALレジスタとオペランド間で乗算が行われる。結果はAL、AHに返される。2バイト乗算の時にはAXレジスタとオペランド間で乗算が行われる。結果はAX、DXに返される 【使用例】　−2×3の場合 MOV AL,0FEH　：AL←−2 MOV BL,3　：BL←+3 IMULBL　：AX←−6
2 2−4 2 2−4	70−77 (76−83)+EA 118−133 (124−139)+EA	X・・????X X・・????X X・・????X X・・????X	MULtiplication unsigned 符号なしの乗算命令 1バイトどうしの乗算と2バイトどうしの乗算が可能 1バイト乗算の時にはALレジスタとオペランド間で乗算が行われる。結果はAL、AHに返される。2バイト乗算の時にはAXレジスタとオペランド間で乗算が行われる。結果はAX、DXに返される 【使用例】　2×3の場合 MOV AL,2　：AL←2 MOV BL,3　：BL←3 MUL BL　：AX←6

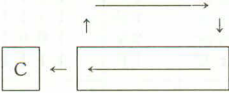
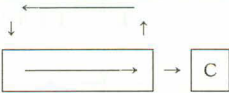
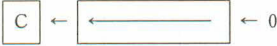
[10-4]　除算命令

ニモニック	オペランド	第1バイト HEX	第1バイト 76543210	第2バイト HEX	第2バイト 76543210
AAD		D5	1 1 0 1 0 1 0 1	0A	0 0 0 0 1 0 1 0
CBW		98	1 0 0 1 1 0 0 0		
CWD		99	1 0 0 1 1 0 0 1		
IDIV	reg 8	F 6	1 1 1 1 0 1 1 0		1 1 1 1 1 r/m
	mem 8	F 6	1 1 1 1 0 1 1 0		mod 1 1 1 r/m
	reg 16	F 7	1 1 1 1 0 1 1 1		1 1 1 1 1 r/m
	mem 16	F 7	1 1 1 1 0 1 1 1		mod 1 1 1 r/m
DIV	reg 8	F 6	1 1 1 1 0 1 1 0		1 1 1 1 0 r/m
	mem 8	F 6	1 1 1 1 0 1 1 0		mod 1 1 0 r/m
	reg 16	F 7	1 1 1 1 0 1 1 1		1 1 1 1 0 r/m
	mem 16	F 7	1 1 1 1 0 1 1 1		mod 1 1 0 r/m

バイト数	クロック数	フラグ ODISZAPC	内　容
2	60	?··XX?X?	Ascii Adjust for Division 除算を行う時にAXレジスタの補正をする。除算命令にはアスキーコード数どうしの命令は用意されていないので、除算を行う前に演算に適した形に補正しなければならない。他の補正命令は演算後に補正するが、この場合は演算前に補正する 【動作】 AL←AH×10+AL　　AH←0 【使用例】　15÷7をアスキーコードで実行する MOV AX,3135H　　：15のアスキーコード MOV BL,37H　　　：7のアスキーコード AND AX,0F0FH　　：AX ← 0105_H AND BL,0FH　　　：BL ← 07_H AAD　　　　　　　：AX ←$000F_H$ DIV　BL　　　　　：AX ← 0102_H
1	2	········	Convert Byte to Word ALレジスタの符号をAHレジスタに拡張する 【動作】 ALレジスタの最上位ビットが0ならAHレジスタを00 Hに、1ならAHレジスタをFFHにする
1	5	········	Convert Word to Doubleword AXレジスタの符号をDXレジスタに拡張する 【動作】 AXレジスタの最上位ビットが0ならDXレジスタを0000 Hに、1ならDXレジスタをFFFFHにする
2 2−4 2 2−4	101−112 (107−118)+EA 165−184 (171−190)+EA	?··????? ?··????? ?··????? ?··?????	Integer DIVision 符号付き除算命令 8ビット長の演算に対しては、AXレジスタに格納されている数値を割る対照とし、商をALレジスタに、余りをAHレジスタに返す 16ビット長の演算に対しては、割られる数値の上位がDXレジスタに、下位はAXレジスタに格納されているものとして演算をほどこし、商はAXレジスタに、余りをDXレジスタに返す 【使用例】　3÷(−2)の場合 MOV　　AX,3 MOV　　BL,FE IDIV　　BL
2 2−4 2 2−4	80−90 (86−96)+EA 144−162 (150−168)+EA	?··????? ?··????? ?··????? ?··?????	DIVision unsigned 符号なし除算命令 8ビット長の演算に対しては、AXレジスタに格納されている数値を割る対象とし、商をALレジスタに、余りをAHレジスタに返す 16ビット長の演算に対しては、割られる数値の上位がDXレジスタに、下位はAXレジスタに格納されているものとして演算をほどこし、商はAXレジスタに、余りをDXレジスタに返す 【使用例】　3÷2の場合 MOV　　AX,3 MOV　　BL,2 DIV　　BL

[10-5] データ転送命令

ニモニック	オペランド	第1バイト		第2バイト	
		HEX	7 6 5 4 3 2 1 0	HEX	7 6 5 4 3 2 1 0
IN	acc,imm 8		1 1 1 0 0 1 0 W		
	acc,DX		1 1 1 0 1 1 0 W		
LAHF		9 F	1 0 0 1 1 1 1 1		
LDS	reg 16, mem 32	C 5	1 1 0 0 0 1 0 1		mod reg r/m
LEA	reg 16, mem 16	8 D	1 0 0 0 1 1 0 1		mod reg r/m
LES	reg 16, mem 32	C 4	1 1 0 0 0 1 0 0		mod reg r/m
MOV	reg,reg		1 0 0 0 1 0 0 W		1 1 reg r/m
	mem,reg		1 0 0 0 1 0 0 W		mod reg r/m
	reg,mem		1 0 0 0 1 0 1 W		mod reg r/m
	mem,imm		1 1 0 0 0 1 1 W		mod 0 0 0 r/m
	reg,imm		1 0 1 1 W reg		
	acc,mem		1 0 1 0 0 0 0 W		
	mem,acc		1 0 1 0 0 0 1 W		
	sreg,reg16	8 E	1 0 0 0 1 1 1 0		1 1 0 sreg r/m
	sreg,mem	8 E	1 0 0 0 1 1 1 0		mod 0 sreg r/m
	reg16,sreg	8 C	1 0 0 0 1 1 0 0		1 1 0 sreg r/m
	mem,sreg	8 C	1 0 0 0 1 1 0 0		mod 0 sreg r/m
OUT	imm 8,acc		1 1 1 0 0 1 1 W		
	DX,acc		1 1 1 0 1 1 1 W		

バイト数	クロック数	フラグ ODISZAPC	内　容
2 1	10 8	・・・・・・・・・ ・・・・・・・・・	INput AL、またはAXレジスタにI/Oポートからのデータを代入する
1	4	・・・・・・・・・	Load AH from Flags フラグレジスタの下位8ビットをAHレジスタに代入する。またAHにフラグデータを取り込めば、フラグデータを一般的なビットデータとして扱うことが可能。そのデータをSAHF命令でフラグレジスタに返すこともできる
2−4	16+EA	・・・・・・・・・	Load pointer using DS 指定したレジスタとDSレジスタに第2オペランドで指定したアドレスから順にデータを取り込む
2−4	2+EA	・・・・・・・・・	Load Effective Addres メモリオペランドによって指定されるオフセット値を指定したレジスタに代入する
2−4	16+EA	・・・・・・・・・	Load pointer using ES 指定したレジスタとESレジスタに第2オペランドで指定したアドレスから順にデータを取り込む
2 2−4 2−4 3−6 2−3 3 3 2 2−4 2 2−4	2 9+EA 8+EA 10+EA 4 10 10 2 8+EA 2 9+EA	・・・・・・・・・ ・・・・・・・・・ ・・・・・・・・・ ・・・・・・・・・ ・・・・・・・・・ ・・・・・・・・・ ・・・・・・・・・ ・・・・・・・・・ ・・・・・・・・・ ・・・・・・・・・ ・・・・・・・・・	MOVe 第2オペランドのデータを第1オペランドへ代入する
2 1	10 8	・・・・・・・・・ ・・・・・・・・・	OUTput I/OポートにALレジスタまたはAXレジスタの値を出力する

ニモニック	オペランド	第1バイト HEX	第1バイト 7 6 5 4 3 2 1	第2バイト HEX	第2バイト 7 6 5 4 3 2 1
POP	mem reg sreg	8F	1 0 0 0 1 1 1 1 0 1 0 1 1 reg 0 0 0 sreg 1 1 1		mod 0 0 0 r/m
POPF		9D	1 0 0 1 1 1 0 1		
PUSH	mem reg sreg	FF	1 1 1 1 1 1 1 1 0 1 0 1 0 reg 0 0 0 sreg 1 1 0		mod 1 1 0 r/m
PUSHF		9C	1 0 0 1 1 1 0 0		
SAHF		9E	1 0 0 1 1 1 1 0		
XCHG	reg, reg mem, reg AX, reg 16		1 0 0 0 0 1 1 W 1 0 0 0 0 1 1 W 1 0 0 1 0 reg		1 1 reg r/m mod reg r/m
XLAT		D7	1 1 0 1 0 1 1 1		

バイト数	クロック数	フラグ ODISZAPC	内 容
2−4 1 1	17+EA 8 8	・・・・・・・・ ・・・・・・・・ ・・・・・・・・	POP word off stack スタックエリアの先頭からデータを取り出しオペランドへ返す
1	8	rrrrrrrr	POP Flags off stack スタックエリアの先頭からフラグレジスタへデータを取り込む
2−4 1 1	16+EA 10 10	・・・・・・・・ ・・・・・・・・ ・・・・・・・・	PUSH word onto stack スタックエリアの先頭にオペランドの内容を書き込む
1	10	・・・・・・・・	PUSH Flags onto stack スタックエリアの先頭にフラグレジスタの内容を書き込む
1	4	・・・XXXXX	Store AH into Flags フラグレジスタの下位8ビットにAHレジスタの値を代入する
2 2−4 1	4 17+EA 3	・・・・・・・・ ・・・・・・・・ ・・・・・・・・	eXCHanGe 第1オペランドと第2オペランドの内容を交換する
1	11	・・・・・・・・	Translate BXレジスタにALレジスタの内容を加算し、この値をオフセット値とし、セグメントをDSレジスタの値で参照されるアドレスからデータを取り出し、ALレジスタに格納する 【使用例】 MOV BX,1000 H MOV [BX],2010 H MOV AL,1 : AL← 1 XLAT : AL← 20_H

[10-6] 論理演算命令

ニモニック	オペランド	第１バイト HEX	第１バイト 7 6 5 4 3 2 1 0	第２バイト HEX	第２バイト 7 6 5 4 3 2 1 0
AND	reg,reg		0 0 1 0 0 0 0 W		1 1 reg r/m
	mem,reg		0 0 1 0 0 0 0 W		mod reg r/m
	reg,mem		0 0 1 0 0 0 1 W		mod reg r/m
	reg,imm		1 0 0 0 0 0 0 W		1 1 1 0 0 r/m
	mem,imm		1 0 0 0 0 0 0 W		mod 1 0 0 r/m
	acc,imm		0 0 1 0 0 1 0 W		
NOT	reg		1 1 1 1 0 1 1 W		1 1 0 1 0 r/m
	mem		1 1 1 1 0 1 1 W		mod 0 1 0 r/m
OR	reg,reg		0 0 0 0 1 0 0 W		1 1 reg r/m
	mem,reg		0 0 0 0 1 0 0 W		mod reg r/m
	reg,mem		0 0 0 0 1 0 1 W		mod reg r/m
	reg,imm		1 0 0 0 0 0 0 W		1 1 0 0 1 r/m
	mem,imm		1 0 0 0 0 0 0 W		mod 0 0 1 r/m
	acc,imm		0 0 0 0 1 1 0 W		
RCL	reg,1		1 1 0 1 0 0 0 W		1 1 0 1 0 r/m
	mem,1		1 1 0 1 0 0 0 W		mod 0 1 0 r/m
	reg,CL		1 1 0 1 0 0 1 W		1 1 0 1 0 r/m
	mem,CL		1 1 0 1 0 0 1 W		mod 0 1 0 r/m
RCR	reg,1		1 1 0 1 0 0 0 W		1 1 0 1 1 r/m
	mem,1		1 1 0 1 0 0 0 W		mod 0 1 1 r/m
	reg,CL		1 1 0 1 0 0 1 W		1 1 0 1 1 r/m
	mem,CL		1 1 0 1 0 0 1 W		mod 0 1 1 r/m

バイト数	クロック数	フラグ ODISZAPC	内　容
2 2−4 2−4 3−4 3−6 2−3	3 16+EA 9+EA 4 17+EA 4	0・・XX?X0 0・・XX?X0 0・・XX?X0 0・・XX?X0 0・・XX?X0 0・・XX?X0	logical AND 第1オペランドと第2オペランドとのANDをとり、結果を第1オペランドに返す
2 2−4	3 16+EA	・・・・・・・・ ・・・・・・・・	logical NOT オペランドの各ビットの反転を行う
2 2−4 2−4 3−4 3−6 2−3	3 16+EA 9+EA 4 17+EA 4	0・・XX?X0 0・・XX?X0 0・・XX?X0 0・・XX?X0 0・・XX?X0 0・・XX?X0	logical OR 第1オペランドと第2オペランドとのORをとり、結果を第1オペランドに返す
2 2−4 2 2−4	2 15+EA 8+4N 20+EA+4N	X・・・・・・X X・・・・・・X ?・・・・・・X ?・・・・・・X	Rotate through Carry Left キャリーと共にメモリまたはレジスタのビットを左へ回転させる。回転する回数は1またはCLレジスタで指定する
2 2−4 2 2−4	2 15+EA 8+4N 20+EA+4N	X・・・・・・X X・・・・・・X ?・・・・・・X ?・・・・・・X	Rotate through Carry Right キャリーと共にメモリまたはレジスタのビットを右へ回転させる。回転する回数は1またはCLレジスタで指定する

ニモニック	オペランド	第1バイト HEX	第1バイト 7 6 5 4 3 2 1 0	第2バイト HEX	第2バイト 7 6 5 4 3 2 1 0
ROL	reg,1 mem,1 reg,CL mem,CL		1 1 0 1 0 0 0 W 1 1 0 1 0 0 0 W 1 1 0 1 0 0 1 W 1 1 0 1 0 0 1 W		1 1 0 0 0 r/m mod 0 0 0 r/m 1 1 0 0 0 r/m mod 0 0 0 r/m
ROR	reg,1 mem,1 reg,CL mem,CL		1 1 0 1 0 0 0 W 1 1 0 1 0 0 0 W 1 1 0 1 0 0 1 W 1 1 0 1 0 0 1 W		1 1 0 0 1 r/m mod 0 0 1 r/m 1 1 0 0 1 r/m mod 0 0 1 r/m
SAL SHL	reg,1 mem,1 reg,CL mem,CL		1 1 0 1 0 0 0 W 1 1 0 1 0 0 0 W 1 1 0 1 0 0 1 W 1 1 0 1 0 0 1 W		1 1 1 0 0 r/m mod 1 0 0 r/m 1 1 1 0 0 r/m mod 1 0 0 r/m
SAR	reg,1 mem,1 reg,CL mem,CL		1 1 0 1 0 0 0 W 1 1 0 1 0 0 0 W 1 1 0 1 0 0 1 W 1 1 0 1 0 0 1 W		1 1 1 1 1 r/m mod 1 1 1 r/m 1 1 1 1 1 r/m mod 1 1 1 r/m
SHR	reg,1 mem,1 reg,CL mem,CL		1 1 0 1 0 0 0 W 1 1 0 1 0 0 0 W 1 1 0 1 0 0 1 W 1 1 0 1 0 0 1 W		1 1 1 0 1 r/m mod 1 0 1 r/m 1 1 1 0 1 r/m mod 1 0 1 r/m
TEST	reg,reg mem,reg reg,imm mem,imm acc,imm		1 0 0 0 0 1 0 W 1 0 0 0 0 1 0 W 1 1 1 1 0 1 1 W 1 1 1 1 0 1 1 W 1 0 1 0 1 0 0 W		1 1 reg r/m mod reg r/m 1 1 0 0 0 r/m mod 0 0 0 r/m
XOR	reg,reg mem,reg reg,mem reg,imm mem,imm acc,imm		0 0 1 1 0 0 0 W 0 0 1 1 0 0 0 W 0 0 1 1 0 0 1 W 1 0 0 0 0 0 0 W 1 0 0 0 0 0 0 W 0 0 1 1 0 1 0 W		1 1 reg r/m mod reg r/m mod reg r/m 1 1 1 1 0 r/m mod 1 1 0 r/m

バイト数	クロック数	フラグ ODISZAPC	内　容
2 2−4 2 2−4	2 15+EA 8+4N 20+EA+4N	X・・・・・・X X・・・・・・X ?・・・・・・X ?・・・・・・X	ROtate Left オペランドのビットを左へ回転させる。回転する回数は1またはCLレジスタで指定 C ← [←] (最上位ビット→C、最上位ビット→最下位ビット)
2 2−4 2 2−4	2 15+EA 8+4N 20+EA+4N	X・・・・・・X X・・・・・・X ?・・・・・・X ?・・・・・・X	ROtate Right オペランドのビットを右へ回転させる。回転する回数は1またはCLレジスタで指定 [→] → C (最下位ビット→C、最下位ビット→最上位ビット)
2 2−4 2 2−4	2 15+EA 8+4N 20+EA+4N	X・・・・・・X X・・・・・・X ?・・・・・・X ?・・・・・・X	Shift Arithmatic Left SHift logical Left オペランドのビットを左へシフトする。シフトする数は1またはCLレジスタで指定 C ← [←] ← 0
2 2−4 2 2−4	2 15+EA 8+4N 20+EA+4N	X・・・・・・X X・・・・・・X ?・・・・・・X ?・・・・・・X	Shift Arithmatic Right オペランドのビットを右へ算術シフトする。シフトする数は1またはCLレジスタで指定 符号→ [→] → C
2 2−4 2 2−4	2 15+EA 8+4N 20+EA+4N	X・・・・・・X X・・・・・・X ?・・・・・・X ?・・・・・・X	SHift logical Right オペランドのビットを右へシフトする。シフトする数は1またはCLレジスタで指定 0 → [→] → C
2 2−4 3−4 3−6 2−3	3 9+EA 5 11+EA 4	0・・XX?X0 0・・XX?X0 0・・XX?X0 0・・XX?X0 0・・XX?X0	TEST 第1オペランドと第2オペランドとのANDをとるが、結果はフラグだけに反映される。オペランドは変化しない
2 2−4 2−4 3−4 3−6 2−3	3 16+EA 9+EA 4 17+EA 4	0・・XX?X0 0・・XX?X0 0・・XX?X0 0・・XX?X0 0・・XX?X0 0・・XX?X0	logcial eXclusive OR 第1オペランドと第2オペランドとのXORをとり、結果を第1オペランドに返す

[10-7] 分岐命令

ニモニック	オペランド	第1バイト HEX	第1バイト 7 6 5 4 3 2 1 0	第2バイト HEX	第2バイト 7 6 5 4 3 2 1 0
CALL	nearproc	E8	1 1 1 0 1 0 0 0		
	regptr 16	FF	1 1 1 1 1 1 1 1		1 1 0 1 0 r/m
	memptr 16	FF	1 1 1 1 1 1 1 1		mod 0 1 0 r/m
	farproc	9A	1 0 0 1 1 0 1 0		
	memptr 32	FF	1 1 1 1 1 1 1 1		mod 0 1 1 r/m
INT	3	CC	1 1 0 0 1 1 0 0		
	imm 8 (≠3)	CD	1 1 0 0 1 1 0 1		
INTO		CE	1 1 0 0 1 1 1 0		
IRET		CF	1 1 0 0 1 1 1 1		
JA JNBE	slabel	77	0 1 1 1 0 1 1 1		
JAE JNB	slabel	73	0 1 1 1 0 0 1 1		
JB JNAE	slabel	72	0 1 1 1 0 0 1 0		
JBE JNA	slabel	76	0 1 1 1 0 1 1 0		
JCXZ	slabel	E3	1 1 1 0 0 0 1 1		

バイト数	クロック数	フラグ ODISZAPC	内　容
3 2 2－4 5 2－4	19 16 21＋EA 28 37＋EA	・・・・・・・・・ ・・・・・・・・・ ・・・・・・・・・ ・・・・・・・・・ ・・・・・・・・・	CALL a procedure オペランドで示されるプロシージャをコールする
1 2	52 51	・・0・・・・・ ・・0・・・・・	INTerrupt 内部割り込み処理を行う時に使われる。セグメント0のはじめの0～3FF$_H$に、ベクタテーブルが用意されており、ジャンプ先を番号によって指定する
1	53/4	・・X・・・・・	INTerrupt if Overflow オーバーフローフラグが1ならタイプ4の割り込みを発生させる
1	24	rrrrrrrr	Interrupt RETurn 割り込み処理からの復帰
2	16/4	・・・・・・・・・	Jump if Above Jump if Not Below nor Equal 上ならジャンプする 【動作】 CF＝0 AND ZF＝0 でジャンプ
2	16/4	・・・・・・・・・	Jump if Above or Equal Jump if Not Below 上か等しければジャンプする 【動作】 CF＝0 でジャンプ
2	16/4	・・・・・・・・・	Jump if Below Jump if Not Above nor Equal 下ならばジャンプする 【動作】 CF＝1 でジャンプ
2	16/4	・・・・・・・・・	Jump if Below or Equal Jump if Not Above 下または等しければジャンプする 【動作】 CF＝1 OR ZF＝1 でジャンプ
2	18/6	・・・・・・・・・	Jump if CX is Zero CXレジスタが0の場合にジャンプする

ニモニック	オペランド	第1バイト HEX	第1バイト 7 6 5 4 3 2 1 0	第2バイト HEX	第2バイト 7 6 5 4 3 2 1 0
JE JZ	slabel	74	0 1 1 1 0 1 0 0		
JG JNLE	slabel	7 F	0 1 1 1 1 1 1 1		
JGE JNL	slabel	7 D	0 1 1 1 1 1 0 1		
JL JNGE	slabel	7 C	0 1 1 1 1 1 0 0		
JLE JNG	slabel	7 E	0 1 1 1 1 1 1 0		
JMP	nlabel slabel regptr 16 memptr 16 flabel memptr 32	E 9 EB FF FF EA FF	1 1 1 0 1 0 0 1 1 1 1 0 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 1 0 1 1 1 1 1 1 1 1		 1 1 1 0 0 r/m mod 1 0 0 r/m mod 1 0 1 r/m
JNE JNZ	slabel	75	0 1 1 1 0 1 0 1		
JNO	slabel	71	0 1 1 1 0 0 0 1		
JNP JPO	slabel	7 B	0 1 1 1 1 0 1 1		

バイト数	クロック数	フラグ ODISZAPC	内　容
2	16/4	・・・・・・・・・	Jump if Equal Jump if Zero 等しければジャンプする 【動作】 ZF＝1 でジャンプ
2	16/4	・・・・・・・・・	Jump if Greater Jump if Not Less nor Equal より大であればジャンプする 【動作】 ZF＝0 AND SF＝OFでジャンプ
2	16/4	・・・・・・・・・	Jump if Greater or Equal Jump if Not Less 以上であればジャンプする 【動作】 SF＝OFでジャンプ
2	16/4	・・・・・・・・・	Jump if Less Jump if Not Greater nor Equal より小であればジャンプする 【動作】 SF≠OFでジャンプ
2	16/4	・・・・・・・・・	Jump if Less nor Equal Jump if Not Greater 以下であればジャンプする 【動作】 ZF＝1 OR SF≠OFでジャンプ
3 2 2 2−4 5 2−4	15 15 11 18＋EA 15 24＋EA	・・・・・・・・・ ・・・・・・・・・ ・・・・・・・・・ ・・・・・・・・・ ・・・・・・・・・ ・・・・・・・・・	JuMP オペランドで示された場所にジャンプする
2	16/4	・・・・・・・・・	Jump if Not Equal Jump if Not Zero 等しくなければジャンプする 【動作】 ZF＝0 でジャンプ
2	16/4	・・・・・・・・・	Jump if Not Overflow オーバーフローでなければジャンプする 【動作】 OF＝0 でジャンプ
2	16/4	・・・・・・・・・	Jump if Not Parity Jump if Parity Odd パリティが奇数ならジャンプする 【動作】 PF＝0 でジャンプ

ニモニック	オペランド	第1バイト		第2バイト	
		HEX	7 6 5 4 3 2 1 0	HEX	7 6 5 4 3 2 1 0
JNS	slabel	79	0 1 1 1 1 0 0 1		
JO	slabel	70	0 1 1 1 0 0 0 0		
JP JPE	slabel	7 A	0 1 1 1 1 0 1 0		
JS	slabel	78	0 1 1 1 1 0 0 0		
LOOP	slabel	E 2	1 1 1 0 0 0 1 0		
LOOPE LOOPZ	slabel slabel	E 1	1 1 1 0 0 0 0 1		

バイト数	クロック数	フラグ ODISZAPC	内　容
2	16/4	・・・・・・・・	Jump if Not Sign サインフラグが0ならジャンプする 【動作】 SF＝0でジャンプ
2	16/4	・・・・・・・・	Jump if Overflow オーバーフローであればジャンプする 【動作】 OF＝1でジャンプ
2	16/4	・・・・・・・・	Jump if Not Parity Jump if Parity Odd パリティが奇数ならジャンプする 【動作】 PF＝1でジャンプ
2	16/4	・・・・・・・・	Jump if Sign サインフラグが1ならジャンプする 【動作】 SF＝1でジャンプ
2	17/5	・・・・・・・・	LOOP CXレジスタから1を引いてCXレジスタが0でなければループする （フラグには影響を与えない） 【動作】 CX＝CX－1 CX≠0であればジャンプする 【使用例】　10から1までの和 　　　　　　MOV　BX,0 　　　　　　MOV　CX,0AH LABEL1：ADD　BX,CX 　　　　　　LOOP　LABEL1
2	18/6	・・・・・・・・	LOOP if Equal LOOP if Zero CXレジスタから1を引いてCXレジスタが0でないか、ゼロフラグが1ならばループする （フラグには影響を与えない） 【動作】 CX＝CX－1 ZF＝1 または CX≠0でループ 【使用例】 ここでは、DS:100$_H$～DS:1FF$_H$までのメモリの内容で、0でないものを見つけたら抜け出すような例を示す 　　　　　　MOV　DI,0 FFH 　　　　　　MOV　CX,100 H LABEL1：INC　DI 　　　　　　CMP　BYTE PTR [DI],0 　　　　　　LOOPE　LABEL 1

ニモニック	オペランド	第1バイト		第2バイト	
		HEX	7 6 5 4 3 2 1 0	HEX	7 6 5 4 3 2 1 0
LOOPNE LOOPNZ	slabel slabel	E 0	1 1 1 0 0 0 0 0		
RET (near) (far)		C 3 CB	1 1 0 0 0 0 1 1 1 1 0 0 1 0 1 1		
RET (near) (far)	pvalue (偶数)	C 2 CA	1 1 0 0 0 0 1 0 1 1 0 0 1 0 1 0		

バイト数	クロック数	フラグ ODISZAPC	内　容
2	19/5	・・・・・・・・・	LOOP if Not Equal LOOP if Not Zero CXレジスタから1を引いてCXレジスタが0でないか、ゼロフラグが0ならばループする （フラグには影響を与えない） 【動作】 CX＝CX-1 ZF＝0 または CX≠0 でループ 【使用例】 ここでは、DS:100$_H$〜DS:1FF$_H$までのメモリの内容で、0を見つけたら抜け出すような例を示す MOV DI,0 FFH MOV CX,100 H LABEL 1: INC DI CMP BYTE PTR [DI],0 LOOPNE LABEL 1
1 1	8 18	・・・・・・・・・ ・・・・・・・・・	RETurn from procedure プロシージャからの復帰
3 3	12 17	・・・・・・・・・ ・・・・・・・・・	RETurn from procedure プロシージャから復帰し、スタックポインタにpvalue値を加算する

[10-8] ストリング命令

ニモニック	オペランド	第1バイト HEX	第1バイト 76543210	第2バイト HEX	第2バイト 76543210
CMPSB CMPSW		A6 A7	10100110 10100111		
LODSB LODSW		AC AD	10101100 10101101		

バイト数	クロック数	フラグ ODISZAPC	内　容
1 1	22 OR 9+22 N 22 OR 9+22 N	X・・XXXXX X・・XXXXX	CoMPare String for Byte CoMPare String for Word セグメントDSレジスタ、オフセットSIレジスタで示されるメモリの内容と、セグメントESレジスタ、オフセットDIレジスタで示されるメモリの内容をバイト（ワード）単位で比較する SI、DIレジスタの値はこの命令の実行後、DF＝0であれば+1（+2）、DF＝1であれば−1（−2）だけ更新される REPZ、REPNZなどと組合せてブロック比較が可能。この時の繰返し数はCXレジスタの値が使われる。CXレジスタは自動的に更新され、CX＝0かREPZやREPNZでの条件が成り立てば実行が終了となる
1 1	12 OR 9+13 N 12 OR 9+13 N	・・・・・・・・ ・・・・・・・・	LOaD String for Byte LOaD String for Word セグメントDSレジスタ、オフセットSIレジスタで示されるメモリの内容を、バイト単位であればALレジスタに、ワード（2バイト）単位であればAXレジスタにロードする SIレジスタの値はこの命令の実行後、DF＝0であれば+1(+2)、DF＝1であれば−1(−2)だけ更新される REP命令と組合せて連続動作が可能となる。この時の繰返し数はCXレジスタの値が使われる。CXレジスタは自動的に更新され、CX＝0になれば実行が終了となる

ニモニック	オペランド	第1バイト		第2バイト	
		HEX	7 6 5 4 3 2 1 0	HEX	7 6 5 4 3 2 1 0
MOVSB MOVSW		A 4 A 5	1 0 1 0 0 1 0 0 1 0 1 0 0 1 0 1		
SCASB SCASW		AE AF	1 0 1 0 1 1 1 0 1 0 1 0 1 1 1 1		
STOSB STOSW		AA AB	1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 1		

バイト数	クロック数	フラグ ODISZAPC	内　容
1 1	18 OR 9+17 N 18 OR 9+17 N	・・・・・・・・・ ・・・・・・・・・	MOVe String for Byte MOVe String for Word セグメントDSレジスタ、オフセットSIレジスタで示されるメモリの内容を、セグメントESレジスタ、オフセットDIレジスタで示されるメモリの内容へバイト（ワード）単位で転送する SI、DIレジスタの値はこの命令の実行後、DF=0であれば+1（+2)、DF=1であれば−1（−2）だけ更新される REP命令と組合せて連続動作が可能となる。この時の繰返し数はCXレジスタの値が使われる。CXレジスタは自動的に更新され、CX=0になれば実行が終了となる
1 1	15 OR 9+15 N 15 OR 9+15 N	X・・XXXXX X・・XXXXX	SCAn String for Byte SCAn String for Word セグメントESレジスタ、オフセットDIレジスタで示されるメモリの内容と、バイト単位であればALレジスタ、ワード（2バイト）単位であればAXレジスタの値との比較を行う DIレジスタの値はこの命令の実行後、DF=0であれば+1(+2)、DF=1であれば−1(−2)だけ更新される REPZ、REPNZなどと組合せて連続比較が可能となる。この時の繰返し数はCXレジスタの値が使われる。CXレジスタは自動的に更新され、CX=0かREPZやREPNZでの条件が成り立てば、実行が終了となる
1 1	11 OR 9+10 N 11 OR 9+10 N	・・・・・・・・・ ・・・・・・・・・	STOre String for Byte STOre String for Word セグメントESレジスタ、オフセットDIレジスタで示されるメモリの内容を、バイト単位であればALレジスタ、ワード（2バイト）単位であればAXレジスタの内容に書き換える DIレジスタの値はこの命令の実行後、DF=0であれば+1(+2)、DF=1であれば−1(−2)だけ更新される REP命令と組合わせて連続動作が可能。繰返し数はCXレジスタの値が使われる。CXレジスタは自動的に更新され、CX=0になれば実行が終了となる

[10-9] ストリング・プリフィックス命令

ニモニック	オペランド	第1バイト HEX	第1バイト 7 6 5 4 3 2 1 0	第2バイト HEX	第2バイト 7 6 5 4 3 2 1 0
REP REPE REPZ		F3	1 1 1 1 0 0 1 1		
REPNE REPNZ		F2	1 1 1 1 0 0 1 0		

[10-10] フラグ制御命令

ニモニック	オペランド	第1バイト HEX	第1バイト 7 6 5 4 3 2 1 0	第2バイト HEX	第2バイト 7 6 5 4 3 2 1 0
CLC		F8	1 1 1 1 1 0 0 0		
CLD		FC	1 1 1 1 1 1 0 0		
CLI		FA	1 1 1 1 1 0 1 0		
CMC		F5	1 1 1 1 0 1 0 1		
STC		F9	1 1 1 1 1 0 0 1		
STD		FD	1 1 1 1 1 1 0 1		
STI		FB	1 1 1 1 1 0 1 1		

バイト数	クロック数	フラグ ODISZAPC	内　容
1	2	・・・・・・・・・	Repeat while CX≠0 Repeat while CX≠0 & ZF=1 Repeat while CX≠0 & ZF=1 ストリング命令の前に置き、次のストリング命令を、CX≠0かZF=1となっている間実行する REP、REPE、REPZはいずれも同じマシン語コードであり、1回の指定のみ有効となっている
1	2	・・・・・・・・・	Repeat while CX≠0 & ZF=0 Repeat while CX≠0 & ZF=0 ストリング命令の前に置き、次のストリング命令を、CX≠0かZF=0となっている間実行する REPNE、REPNZはいずれも同じマシン語コードであり、1回の指定のみ有効となっている

バイト数	クロック数	フラグ ODISZAPC	内　容	
1	2	・・・・・・・・0	CLear Carry flage キャリーフラグをクリア	CF=0
1	2	・0・・・・・・	CLear Direction flage ディレクションフラグをクリア	DF=0
1	2	・・0・・・・・	CLear Interrupt flage 割り込みフラグをクリア	IF=0
1	2	・・・・・・・X	CoMplement Carry flage キャリーフラグの反転を行う	$CF=\overline{CF}$
1	2	・・・・・・・1	SeT Carry flage キャリーフラグをセットする	CF=1
1	2	・1・・・・・・	SeT Direction flage ディレクションフラグをセットする	DF=1
1	2	・・1・・・・・	SeT Interrupt flage 割り込みフラグをセットする	IF=1

[10-11] CPU制御命令

ニモニック	オペランド	第1バイト		第2バイト	
		HEX	7 6 5 4 3 2 1 0	HEX	7 6 5 4 3 2 1 0
ESC	exop,reg exop,mem		1 1 0 1 1 X X X 1 1 0 1 1 X X X		1 1 Y Y Y r/m mod Y Y Y r/m
HLT		F 4	1 1 1 1 0 1 0 0		
LOCK		F 0	1 1 1 1 0 0 0 0		
NOP		90	1 0 0 1 0 0 0 0		
WAIT		9 B	1 0 0 1 1 0 1 1		

[10-12] セグメント・オーバーライド命令

ニモニック	オペランド	第1バイト		第2バイト	
		HEX	7 6 5 4 3 2 1 0	HEX	7 6 5 4 3 2 1 0
CS:		2 E	0 0 1 0 1 1 1 0		
DS:		3 E	0 0 1 1 1 1 1 0		
ES:		26	0 0 1 0 0 1 1 0		
SS:		36	0 0 1 1 0 1 1 0		

バイト数	クロック数	フラグ ODISZAPC	内 容
2 2−4	2 8+EA	･････････ ･････････	ESCape データバスにオペランドで指定したメモリの内容を設定する
1	2	･････････	HaLT CPUの実行停止命令
1	2	･････････	LOCK bus バスのロック信号を設定する
1	3	･････････	No OPeration 何も実行しない命令
1	3+5N	･････････	Wait testピンの信号がアクティブになるまでウエイトする

バイト数	クロック数	フラグ ODISZAPC	内 容
1	2	･････････	セグメントの参照をCSレジスタとする
1	2	･････････	セグメントの参照をDSレジスタとする
1	2	･････････	セグメントの参照をESレジスタとする
1	2	･････････	セグメントの参照をSSレジスタとする

アスキーコード一覧表

上位4ビット→

下位4ビット↓

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0		DE	SP	0	@	P	‘	p				ー	タ	ミ		
1	SH	D1	!	1	A	Q	a	q			。	ア	チ	ム		円
2	SX	D2	"	2	B	R	b	r			「	イ	ツ	メ		年
3	EX	D3	#	3	C	S	c	s			」	ウ	テ	モ		月
4	ET	D4	$	4	D	T	d	t			、	エ	ト	ヤ		日
5	EQ	NK	%	5	E	U	e	u			・	オ	ナ	ユ		時
6	AK	SN	&	6	F	V	f	v			ヲ	カ	ニ	ヨ		分
7	BL	EB	'	7	G	W	g	w			ァ	キ	ヌ	ラ		秒
8	BS	CN	(	8	H	X	h	x			ィ	ク	ネ	リ	♠	
9	HT	EM	)	9	I	Y	i	y			ゥ	ケ	ノ	ル	♥	
A	LF	SB	*	:	J	Z	j	z			ェ	コ	ハ	レ	♦	
B	HM	EC	+	;	K	[	k	{			ォ	サ	ヒ	ロ	♣	
C	CL	→	,	<	L	¥	l	¦			ャ	シ	フ	ワ	●	
D	CR	←	−	=	M	]	m	}			ュ	ス	ヘ	ン	○	
E	SO	↑	.	>	N	^	n	~			ョ	セ	ホ	゛		
F	SI	↓	/	?	O	_	o				ッ	ソ	マ	゜		

(注) SP は空白(スペース)コードを示します

この『8086マシン語秘伝の書』をまとめるにあたり、以下の文献にお世話になりました。この場をかりてお礼を申しあげます。

◆本　文

『ザ8086ブック』吉川敏則 訳　産報出版
『PC-9801マシン語入門』岩瀬正幸・藤井敬雄 共著　アスキー
『はじめて読むMASM』蒲地輝尚 著　アスキー
『MS-DOSプログラミングテクニック』アスキー書籍編集部 編著　アスキー
『8086アセンブリ言語』西村義考 著　日本ソフトバンク
『8086プログラミングデザイン』山内　直 著　秀和システムトレーディング
『FORTRANのための数値計算法』村越勝弘 訳　科学技術出版社
『マシン語クックブック1』藤田英時・幸田敏記 共著　システムソフト
『PC-9800はじめてのマシン語』日高徹・青山学 共著　啓学出版

◆ニモニック表

『ザ8086ブック』吉川敏則 訳　産報出版
『PC-9801マシン語入門』岩瀬正幸・藤井敬雄 共著　アスキー
『8086アセンブリ言語』西村義考 著　日本ソフトバンク
『8086プログラミングデザイン』山内　直 著　秀和システムトレーディング

<著者紹介>

日高　徹

1949 年　栃木県宇都宮市生まれ

早稲田大学商学部卒業後、商社やカー用品メーカーに十数年勤務。現在はフリーのゲームデザイナー。
主な作品は、『ホーンテッドケイブ』『マジックガーデン』『ガンダーラ』など。著書に『マシン語ゲームプログラミング』『PC-8801 シリーズ マシン語サウンド・プログラミング』(以上アスキー)『PC-8801 シリーズ マシン語ゲームグラフィックス』(小学館)『PC-8800 シリーズ はじめてのマシン語』『PC-9800 シリーズ はじめてのマシン語』『Z 80 マシン語秘伝の書』(以上啓学出版) がある。
趣味はトレーニング。剣道三段。運転免許証を全種類もつ隠れプロドライバーでもある。

青山　学

1958 年　東京生まれ

青山学院大学理工学部卒業後、コンピュータエンジニアを経て、現在はフリーのゲームデザイナー。
主な作品は、98 版『ゼビウス』、88 版『ドラゴンバスター』など。
趣味はスキー、テニス。

[編集部からのお願い]

本書の内容に関する質問等は書面にて当編集部宛へお送り願います。電話による質問等にはいっさい応じられません。

8086 マシン語秘伝の書

1990年10月31日　第 1 刷発行
1991年 5 月31日　第 2 刷発行

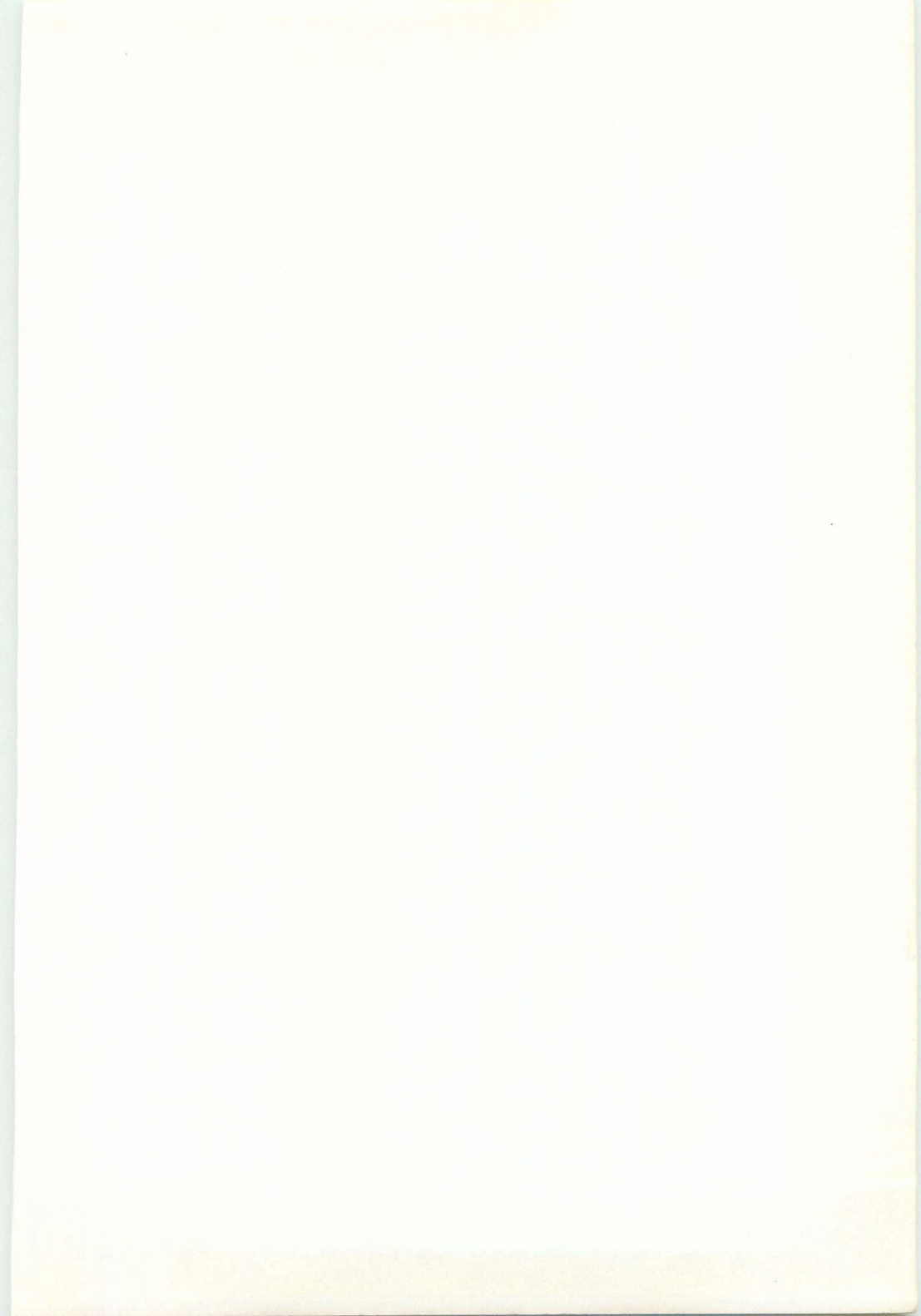
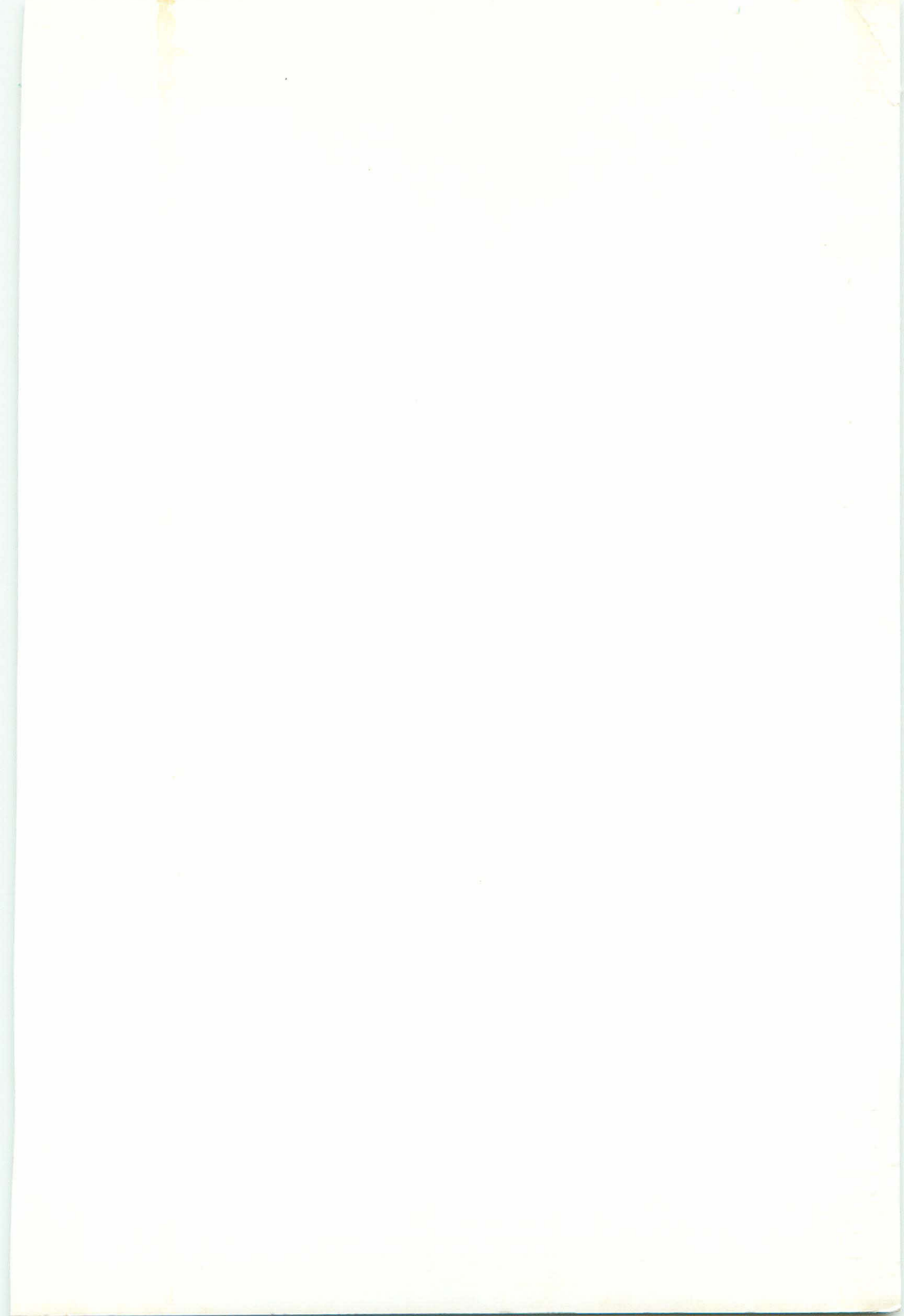
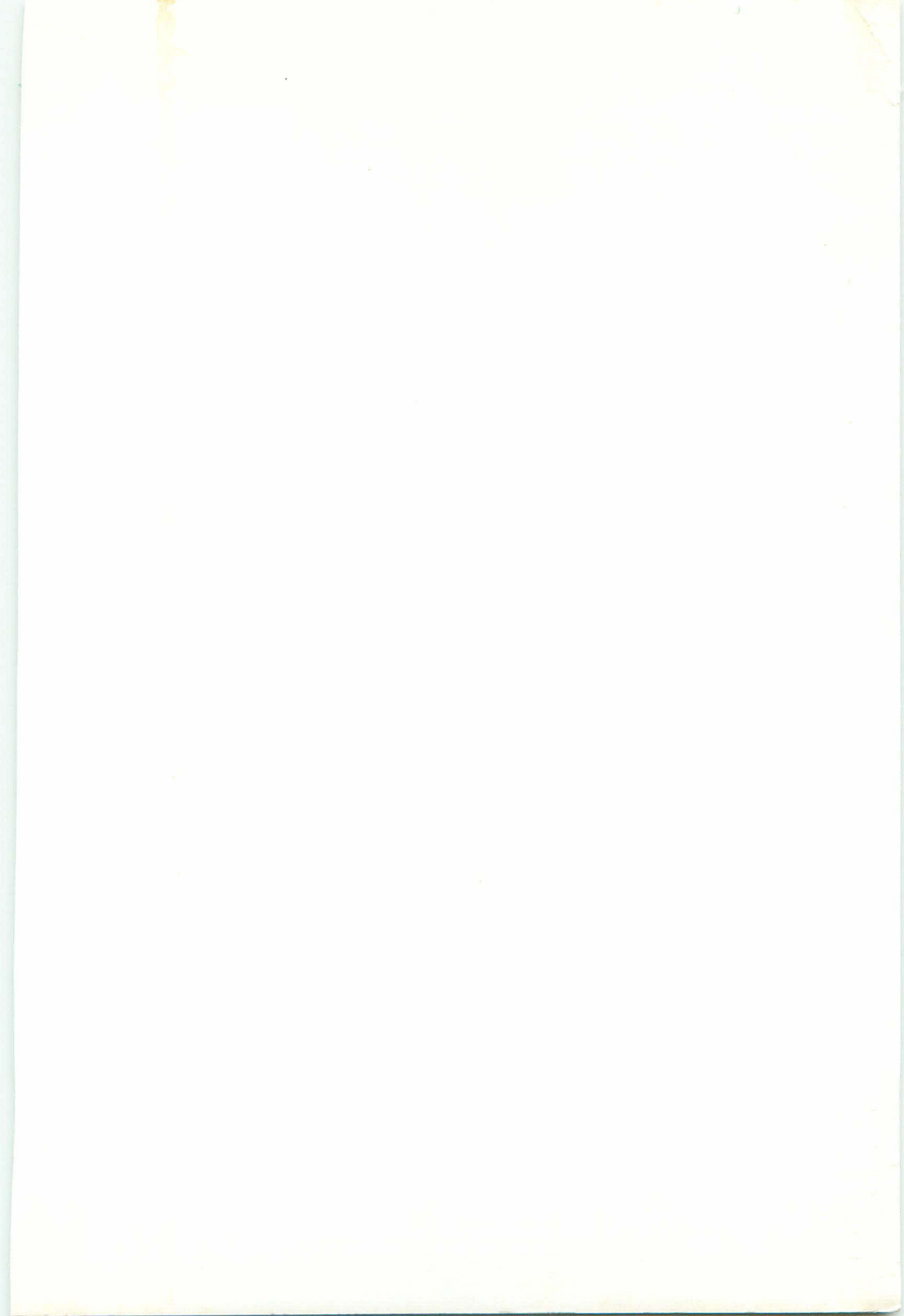
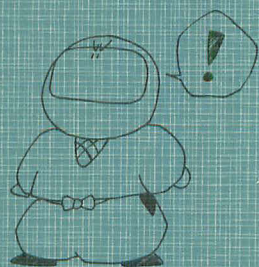
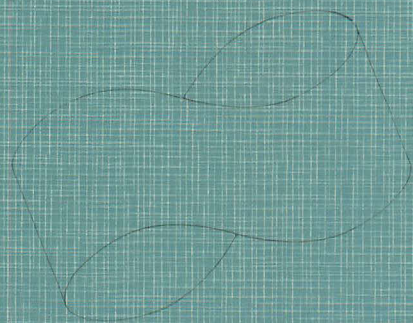
著　者　日高(ひだか)　徹(とおる)
　　　　青山(あおやま)　学(まなぶ)

発行所　啓学出版株式会社
代表者　三井数美

郵便番号　101
東京都千代田区神田神保町1-46
電話　東京03(3233)3731[編集部]
　　　東京03(3233)3795[販売部]
振替　東京 3-109286

印刷／昭和工業写真印刷所
製本／徳住製本所

ISBN 4-7665-1079-8
本書の定価はカバーに表示してあります

Printed in Japan
Tsuchi

8086マシン語秘伝の書

啓学出版
(出)1953

ISBN 4-7665-1079-8 C3055 P2000E
定価2,000円(本体1,942円)

8086マシン語秘伝の書

(出)1953